FWX

Future Trading

Smart Contract Audit Report

Date Issued: 30 Sep 2024

Version: Final v1.1

Public

Table of Contents

Executive Summary

Overview

Valix conducted a smart contract audit to evaluate potential security issues of the **Future Trading**. This audit report was published on `30 Sep 2024`. The audit scope is limited to the **Future Trading.** Our security best practices strongly recommend that the **FWX team** conduct a full security audit for both on-chain and off-chain components of its infrastructure and their interaction. A comprehensive examination has been performed during the audit process utilizing Valix's Formal Verification, Static Analysis, and Manual Review techniques.

About Future Trading

Conditional Orders

Definition: A Conditional Order is a limit order that allows you to open a position at a specific price you are interested in. It enables you to enter the market only when the asset reaches your desired price level.

How It Works: To place a Conditional Order, you need to provide the required collateral. An operational fee of $0.50 USD is charged per order when the current market price reaches your target price and the order is executed. You can set up to a maximum of 5 Conditional Orders at a time.

Setting Target Prices:

For Long Positions: When opening a long position, your target price must be lower than the current market price. This means you aim to buy the asset at a price below its current level, anticipating that the price will rise in the future.

For Short Positions: When opening a short position, your target price must be higher than the current market price. This allows you to sell the asset at a higher price, expecting that its value will decrease over time.

Take Profit (TP) / Stop Loss (SL)

Definition: TP/SL orders enable you to close an existing position at a specific price level you choose. You can set TP/SL orders only after the position has been opened.

How It Works: An operational fee of $0.50 USD is charged after the position is closed using a TP or SL order. You can set up to a maximum of five (5) TP/SL orders at a time.

Setting Target Prices:

For Long Positions:

Take Profit (TP): Set a TP price that is higher than the current market price to lock in profits when the price

rises to your target level.
Stop Loss (SL): Set an SL price that is lower than the current market price to limit potential losses if the price drops below your specified level.

For Short Positions:
Take Profit (TP): Set a TP price that is lower than the current market price to realize profits when the price decreases to your target level.
Stop Loss (SL): Set an SL price that is higher than the current market price to mitigate losses if the price increases beyond your specified level.

Scope of Work

The security audit conducted does not replace the full security audit of the overall **FWX** protocol. The scope is limited to the **Future Trading** and their related smart contracts.

The security audit covered the components at this specific state:

Item	Description
Components	▪ *Conditional smart contracts* ▪ *Core smart contracts* ▪ *Factory smart contracts* ▪ *NFT smart contracts* ▪ *Pool smart contracts* ▪ *Stakepool smart contracts* ▪ *SmartWallet smart contracts* ▪ *Forwarder smart contracts* ▪ *Imported associated smart contracts and libraries*
Git Repository	▪ *https://github.com/forward-x/defi-protocol-future-trading*
Audit Commit	▪ *26f011369e3f4ba415e158b7f59dd4c764184eed (branch: develop-v-1-4)*
Certified Commit	▪ *ae3e011cf28285bd79fe16f4d5336587ef598601 (branch: develop-v-1-4)*
Audited Files	▪ *contracts/src/conditional/event/ConditionalEvent.sol* ▪ *contracts/src/conditional/Conditional.sol* ▪ *contracts/src/conditional/ConditionalBase.sol* ▪ *contracts/src/conditional/ConditionalFunc.sol*

	▪ *contracts/src/conditional/ConditionalOrder.sol* ▪ *contracts/src/conditional/ConditionalProxy.sol* ▪ *contracts/src/conditional/ConditionalTmpStruct.sol* ▪ *contracts/src/conditional/ConditionalTPSL.sol* ▪ *contracts/src/core/event/CoreBorrowingEvent.sol* ▪ *contracts/src/core/event/CoreEvent.sol* ▪ *contracts/src/core/event/CoreFutureTradingEvent.sol* ▪ *contracts/src/core/event/CoreSettingEvent.sol* ▪ *contracts/src/core/event/FeeVaultEvent.sol* ▪ *contracts/src/core/APHCore.sol* ▪ *contracts/src/core/APHCoreProxy.sol* ▪ *contracts/src/core/APHCoreSettingProxy.sol* ▪ *contracts/src/core/APHCoreV2.sol* ▪ *contracts/src/core/CoreBase.sol* ▪ *contracts/src/core/CoreBaseFunc.sol* ▪ *contracts/src/core/CoreBorrowing.sol* ▪ *contracts/src/core/CoreFutureBaseFunc.sol* ▪ *contracts/src/core/CoreFutureClosing.sol* ▪ *contracts/src/core/CoreFutureOpening.sol* ▪ *contracts/src/core/CoreFutureWallet.sol* ▪ *contracts/src/core/CoreLiquidateWithoutSwap.sol* ▪ *contracts/src/core/CoreSetting.sol* ▪ *contracts/src/core/CoreSwappingV3.sol* ▪ *contracts/src/core/CoreTmpStruct.sol* ▪ *contracts/src/core/FeeVault.sol* ▪ *contracts/src/gasless/event/ForwarderEvent.sol* ▪ *contracts/src/gasless/event/SmartWalletEvent.sol* ▪ *contracts/src/gasless/Forwarder.sol* ▪ *contracts/src/gasless/ForwarderBase.sol* ▪ *contracts/src/gasless/ForwarderConditionalTrading.sol* ▪ *contracts/src/gasless/ForwarderCore.sol* ▪ *contracts/src/gasless/ForwarderFunc.sol* ▪ *contracts/src/gasless/ForwarderMembership.sol* ▪ *contracts/src/gasless/ForwarderPool.sol* ▪ *contracts/src/gasless/ForwarderStruct.sol* ▪ *contracts/src/gasless/SmartWallet.sol* ▪ *contracts/src/gasless/SmartWalletFactory.sol* ▪ *contracts/src/gasless/Verifier.sol* ▪ *contracts/src/gasless/VerifierBase.sol* ▪ *contracts/src/gasless/VerifierFunc.sol* ▪ *contracts/src/gasless/VerifierStruct.sol* ▪ *contracts/src/libraries/APHLibrary.sol* ▪ *contracts/src/nft/Membership.sol* ▪ *contracts/src/pool/event/InterestVaultEvent.sol* ▪ *contracts/src/pool/event/PoolBorrowingEvent.sol* ▪ *contracts/src/pool/event/PoolLendingEvent.sol* ▪ *contracts/src/pool/event/PoolSettingEvent.sol* ▪ *contracts/src/pool/APHPool.sol* ▪ *contracts/src/pool/APHPoolProxy.sol* ▪ *contracts/src/pool/APHPoolV2.sol* ▪ *contracts/src/pool/InterestVault.sol*

	▪ contracts/src/pool/InterestVaultV2.sol ▪ contracts/src/pool/PoolBase.sol ▪ contracts/src/pool/PoolBaseFunc.sol ▪ contracts/src/pool/PoolBorrowing.sol ▪ contracts/src/pool/PoolLending.sol ▪ contracts/src/pool/PoolLendingV2.sol ▪ contracts/src/pool/PoolSetting.sol ▪ contracts/src/pool/PoolToken.sol ▪ contracts/src/stakepool/StakePool.sol ▪ contracts/src/stakepool/StakePoolBase.sol ▪ contracts/src/stakepoolV2/StakePoolBaseV2.sol ▪ contracts/src/stakepoolV2/StakePoolSettingV2.sol ▪ contracts/src/stakepoolV2/StakePoolV2.sol ▪ contracts/src/utils/PriceFeed.sol ▪ contracts/src/utils/PriceFeedL2.sol ▪ contracts/src/utils/Vault.sol ▪ contracts/src/utils/WETHHandler.sol
Excluded Files/Contracts	▪ contracts/examples/*.sol ▪ contracts/externalContract/forwardswap/*.sol ▪ contracts/externalContract/multicall/*.sol ▪ contracts/externalContract/pancake/*.sol ▪ contracts/externalContract/uniswapv3/*.sol ▪ contracts/mock/*.sol ▪ contracts/src/helper/*.sol ▪ contracts/src/core/CoreSwapping.sol ▪ contracts/src/core/CoreSwappingV2.sol ▪ contracts/src/core/CoreSwappingUniV3.sol ▪ contracts/src/utils/eFWX.sol

Remark: Our security best practices strongly recommend that the FWX team conduct a full security audit for both on-chain and off-chain components of its infrastructure and the interaction between them.

Auditors

Role	Staff List
Auditors	**Anak Mirasing** **Kritsada Dechawattana** **Parichaya Thanawuthikrai** **Nattawat Songsom**
Authors	**Anak Mirasing** **Kritsada Dechawattana** **Parichaya Thanawuthikrai** **Nattawat Songsom**
Reviewers	**Sumedt Jitpukdebodin**

Disclaimer

Our smart contract audit was conducted over a limited period and was performed on the smart contract at a single point in time. As such, the scope was limited to current known risks during the work period. The review does not indicate that the smart contract and blockchain software has no vulnerability exposure.

We reviewed the security of the smart contracts with our best effort, and we do not guarantee a hundred percent coverage of the underlying risk existing in the ecosystem. The audit was scoped only in the provided code repository. The on-chain code is not in the scope of auditing.

This audit report does not provide any warranty or guarantee, nor should it be considered an “approval” or “endorsement” of any particular project. This audit report should also not be used as investment advice nor provide any legal compliance.

Audit Result Summary

From the audit results and the remediation and response from the developer, Valix trusts that the **Future Trading** have sufficient security protections to be safe for use.

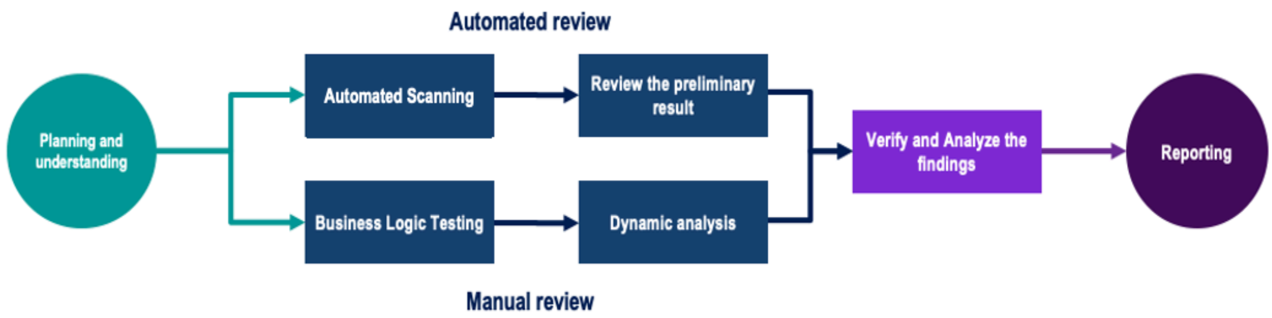
Initially, Valix was able to identify **38 issues** that were categorized from the “Critical” to “Informational” risk level in the given timeframe of the assessment. **Of these, the team was able to completely fix 27 issues and acknowledged 11 issues**. Below is the breakdown of the vulnerabilities found and their associated risk rating for each assessment conducted.

Target	Assessment Result					Reassessment Result				
	C	H	M	L	I	C	H	M	L	I
Future Trading	2	12	14	4	6	0	3	7	0	1

Note: *Risk Rating* **C** *Critical,* **H** *High,* **M** *Medium,* **L** *Low,* **I** *Informational*

Methodology

The smart contract security audit methodology is based on Smart Contract Weakness Classification and Test Cases (SWC Registry), CWE, well-known best practices, and smart contract hacking case studies. Manual and automated review approaches can be mixed and matched, including business logic analysis in terms of the malicious doer's perspective. Using automated scanning tools to navigate or find offending software patterns in the codebase along with a purely manual or semi-automated approach, where the analyst primarily relies on one's knowledge, is performed to eliminate the false-positive results.



Planning and Understanding

- Determine the scope of testing and understanding of the application’s purposes and workflows.
- Identify key risk areas, including technical and business risks.
- Determine which sections to review within the resource constraints and review method – automated, manual or mixed.

Automated Review

- Adjust automated source code review tools to inspect the code for known unsafe coding patterns.
- Verify the tool's output to eliminate false-positive results, and adjust and re-run the code review tool if necessary.

Manual Review

- Analyzing the business logic flaws requires thinking in unconventional methods.
- Identify unsafe coding behavior via static code analysis.

Reporting

- Analyze the root cause of the flaws.
- Recommend improvements for secure source code.

Audit Items

We perform the audit according to the following categories and test names.

Category	ID	Test Name
Security Issue	SEC01	*Authorization Through tx.origin*
	SEC02	*Business Logic Flaw*
	SEC03	*Delegatecall to Untrusted Callee*
	SEC04	*DoS With Block Gas Limit*
	SEC05	*DoS with Failed Call*
	SEC06	*Function Default Visibility*
	SEC07	*Hash Collisions With Multiple Variable Length Arguments*
	SEC08	*Incorrect Constructor Name*
	SEC09	*Improper Access Control or Authorization*
	SEC10	*Improper Emergency Response Mechanism*
	SEC11	*Insufficient Validation of Address Length*
	SEC12	*Integer Overflow and Underflow*
	SEC13	*Outdated Compiler Version*
	SEC14	*Outdated Library Version*
	SEC15	*Private Data On-Chain*
	SEC16	*Reentrancy*
	SEC17	*Transaction Order Dependence*
	SEC18	*Unchecked Call Return Value*
	SEC19	*Unexpected Token Balance*
	SEC20	*Unprotected Assignment of Ownership*
	SEC21	*Unprotected SELFDESTRUCT Instruction*
	SEC22	*Unprotected Token Withdrawal*
	SEC23	*Unsafe Type Inference*
	SEC24	*Use of Deprecated Solidity Functions*
	SEC25	*Use of Untrusted Code or Libraries*
	SEC26	*Weak Sources of Randomness from Chain Attributes*
	SEC27	*Write to Arbitrary Storage Location*

Category	ID	Test Name
Functional Issue	**FNC01**	*Arithmetic Precision*
	FNC02	*Permanently Locked Fund*
	FNC03	*Redundant Fallback Function*
	FNC04	*Timestamp Dependence*
Operational Issue	**OPT01**	*Code With No Effects*
	OPT02	*Message Call with Hardcoded Gas Amount*
	OPT03	*The Implementation Contract Flow or Value and the Document is Mismatched*
	OPT04	*The Usage of Excessive Byte Array*
	OPT05	*Unenforced Timelock on An Upgradeable Proxy Contract*
Developmental Issue	**DEV01**	*Assert Violation*
	DEV02	*Other Compilation Warnings*
	DEV03	*Presence of Unused Variables*
	DEV04	*Shadowing State Variables*
	DEV05	*State Variable Default Visibility*
	DEV06	*Typographical Error*
	DEV07	*Uninitialized Storage Pointer*
	DEV08	*Violation of Solidity Coding Convention*
	DEV09	*Violation of Token (ERC20) Standard API*

Risk Rating

To prioritize the vulnerabilities, we have adopted the scheme of five distinct levels of risk: **Critical**, **High**, **Medium**, **Low**, and **Informational**, based on OWASP Risk Rating Methodology. The risk level definitions are presented in the table.

Risk Level	Definition
Critical	The code implementation does not match the specification, and it could disrupt the platform.
High	The code implementation does not match the specification, or it could result in losing funds for contract owners or users.
Medium	The code implementation does not match the specification under certain conditions, or it could affect the security standard by losing access control.
Low	The code implementation does not follow best practices or use suboptimal design patterns, which may lead to security vulnerabilities further down the line.
Informational	Findings in this category are informational and may be further improved by following best practices and guidelines.

The **risk value** of each issue was calculated from the product of the **impact** and **likelihood values**, as illustrated in a two-dimensional matrix below.

- **Likelihood** represents how likely a particular vulnerability is exposed and exploited in the wild.
- **Impact** measures the technical loss and business damage of a successful attack.
- **Risk** demonstrates the overall criticality of the risk.

Impact \ Likelihood	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Informational

The shading of the matrix visualizes the different risk levels. Based on the acceptance criteria, the risk levels "Critical" and "High" are unacceptable. Any issue obtaining the above levels must be resolved to lower the risk to an acceptable level.

Findings

System Trust Assumptions

Trust assumptions

The trust assumption in this context is that the **Future Trading** protocol allows the trusted operator to oversee the protocol.

It's important to note that, while the trusted operator is granted specific privileges to oversee the **Future Trading** protocol, special attention should be given to the account with the ***addressTimelockManager***, ***noTimelockManager***, ***configTimelockManager*** and **owner** role. These accounts have the authority to change the address of external calls, pause functionalities and change protocol configuration.

Furthermore, the trusted operator can execute actions without the need for a time-lock mechanism. This implies that any action within the scope of the trusted operator's authority will be carried out promptly.

The privileged roles

In the **Future Trading** protocol, privileged roles have special access to perform sensitive actions, relying on the trust placed in these roles to ensure the proper functioning and security of the system.

The **Conditional** contract:

- The **noTimelockManager** account:
    - Can pause and unpause several functionalities such as setting TPSL orders etc.
- The **addressTimelockManager** account:
    - Can set the address of the *Core* contract.
    - Can set the address of the *HelperFuture* contract.
    - Can set the length limit of new TPSL orders.
    - Can set the length limit of new conditional orders.
    - Can set the address of the *ConditionalTPSL* contract.
    - Can set the address of the *ConditionalOrder* contract.
- The **configTimelockManager** account:
    - Can set the service charge of triggering conditional orders.
    - Can set the service charge of triggering TPSL orders.

  - Can approve Conditional contract's tokens to the *noTimelockManager* account.

The **APHCore** contract:

- The **addressTimelockManager** account:
  - Can set the address of the *CoreSetting* contract.
- The **noTimelockManager** account:
  - Can pause and unpause several functionalities such as opening future positions etc.

The **CoreSetting, (APHCoreSetting still not inconsistent)** contract:

- The **addressTimelockManager** account:
  - Can set the address of the *Membership* contract.
  - Can set the address of the *PriceFeed* contract.
  - Can set the address of the *ForwLendingVault* contract.
  - Can set the address of the *FORWTradingVault* contract.
  - Can set the address of the *WETHHandler* contract.
  - Can set the address of the *CoreBorrowing* contract.
  - Can set the address of the *CoreFutureWallet* contract.
  - Can set the address of the *CoreFutureOpening* contract.
  - Can set the address of the *CoreFutureClosing* contract.
  - Can set the address of the *CoreSwapping* contract.
  - Can set the address of the *CoreLiquidateWithoutSwap* contract.
  - Can set the address of the *FeeVault* contract.
  - Can set the address of the *UniswapV3 quoter* contract.
  - Can set the address of *Conditional* contract.
  - Can approve tokens to each DEX routers.
  - Can set the whitelisted tokens to be used as future position collateral.
  - Can set address DEX routers to interact with.
- The **configTimelockManager** account:

- Can set the period of loan that does not need to be rollovered.
- Can set the minimum interest duration.
- Can set the percentage of future position liquidation fee.
- Can set the loan liquidation fee.
- Can set the maximum leverage allowed for future positions.
- Can set the percentage of interest to be splitted as *heldTokenInterest*.
- Can set the percentage of trading fee to lenders.
- Can set the percentage of auction spread.
- Can set the configuration of future positions such as minimum/maintenance margin, bounty fee to liquidator and protocol, minimum/maximum position size.
- Can register new *APHPools*.
- Can set the configuration to distribute *FORW* tokens to each *APHPool*.
- Can set the configuration to get *FORW* token bonus such as *FORW* bonus amount, target position size to get the bonus.
- Can set the configuration of interaction with DEX routers such as max swap size, max price impact, max price different percent from the oracle.
- Can set the swap fee rate of each DEX routers.
- Can set the swap fee rate of each token pair in DEX routers.
- Can set the value of *forwStakingMultiplier* which will be used to determine whether the staking balance is enough to deposit more tokens.
- Can set the address of the *Forwarder* contract.

The **APHCore** contract:

- The **addressTimelockManager** account:
  - Can set the address of the *CoreSetting* contract.
- The **noTimelockManager** account:
  - Can pause and unpause several functionalities such as borrowing tokens etc.

The **FeeVault** contract:

- The **addressTimelockManager** account:
    - Can set the address of the *Core* contract, this contract is allowed to add profit fee and auction fee to *FeeVault*.
    - Can set the address of the auction fee receiver contract.
    - Can set the address of the profit fee receiver account.

The **Membership** contract:

- The **addressTimelockManager** account:
    - Can set the address of the *stakePool* contract.
- The **configTimelockManager** account:
    - Can set NFT base token URI.
- The **noTimelockManager** account:
    - Can pause and unpause several functionalities such as setting the address of the *stakePool* contract etc.

The **APHPool** contract:

- The **noTimelockManager** account:
    - Can pause and unpause several functionalities such as borrowing tokens etc.

The **APHPoolV2** contract:

- The **noTimelockManager** account:
    - Can pause and unpause several functionalities such as borrowing tokens etc.

The **InterestVault, InterestVaultV2** contract:

- The **addressTimelockManager** account:
    - Can set the address of the *FORW* token contract.
    - Can set the address of the profit token contract.
    - Can set the address of the treasury account.
    - Can set the address of the *Core* contract, this contract is allowed to call *settleInterest* to settle the value of claimable token interest and held token interest.
    - Can approve tokens of *InterestVault* contract.

- The **noTimelockManager** account:
  - Can withdraw *actualTokenInterestProfit.*
- The **owner** account:
  - Can call *withdrawTokenInterest* to subtract token interest value and add actual profit.
  - Can call *withdrawForwInterest* to subtract *forw* interest value.

The **PoolSetting** contract:

- The **onlyConfigTimelockManager** account:
  - Can set borrow interest rate and utilization rate configurations.
  - Can set loan configurations including allowed collateral token address, safe LTV for LTV configuration, max LTV allowed for borrowing, liquidate LTV, percentage of *bountyFeeRate* for liquidators and protocol, *penaltyFeeRate* for *rollover* operation, *maxOraclePriceDiffPercent* for borrowing and liquidating, minimum amount of collateral.
  - Can set the lambda value, this value is used as weight in *ifp* token price calculation.

The **StakePool, StakePoolSettingV2** contract:

- The **onlyAddressTimelockManager** account:
  - Can set the address of the next *StakePool* contract, this new *StakePool* contract is allowed to deprecate user stake information.
- The **onlyConfigTimelockManager** account:
  - Can set the stake settle interval
  - Can set stake start time.
  - Can set the benefits of each NFT rank including *interestBonusLending*, *forwardBonusLending*, *minimumStakeAmount*, *maxLTVBonus*, *tradingFee* and *tradingBonus*.
  - Can pause and unpause several functionalities such as staking tokens etc.

The **LiquidationCenter** contract

- The **owner** account:
  - Can withdraw tokens from the *LiquidationCenter* contract.
  - Can set the address of the *Core* contract.

The **PriceFeed** and **PriceFeedL2** contract:

- The **configTimelockManager** account:
  - Can set the address of the external *PriceFeed* Oracle contract and decimal value for several tokens.
  - Can set the acceptable stale period for several tokens.
- The **noTimelockManager** account:
  - Can pause and unpause the query USD price rate functionality.

The **Forwarder** contract:

- The **addressTimelockManager** account:
  - Can withdraw the tokens in the *Forwarder* contract to a specified recipient.
  - Can approve the specified address to spend a certain amount of tokens on behalf of the *Forwarder* contract.
  - Can set the address of the *executionChargeVaultAddress*.
  - Can approve the executor's whitelist address to allow it to execute the request.
  - Can approve the execution whitelist token.
  - Can set the address of the *APHCore* contract.
  - Can set the address of the *Conditional* contract.
  - Can set the address of the *Verifier* contract.
  - Can set the address of the *SmartWalletFactory* contract
  - Can set the execution rate for specific tokens.

Review Findings Summary

The table below shows the summary of our assessments.

No.	Issue	Risk	Status	Functionality is in use
1	Loss Tracking Precision Mismatch In APHCore	Critical	Fixed	In use
2	Unable Migration For StakePool To StakePoolV2	Critical	Fixed	In use
3	Inconsistent Calculation For The Liquidation Fee	High	Fixed	In use
4	Position Order Opens Out Of Slippage	High	Acknowledged	In use
5	Potential Withdraw Collateral Even Reach The Liquidation Through The adjustCollateral Function	High	Fixed	In use
6	Misclassification Of Forwarder As Liquidator In Repay And Rollover Functions	High	Acknowledged	In use
7	Incorrect Bounty Fee Distribution In Liquidation Scenarios	High	Acknowledged	In use
8	Service Charge Lockup In Forwarder Contract Due To setTPSL Function	High	Fixed	In use
9	Incompatibility Issues In PoolLendingV2	High	Fixed	In use
10	Incorrect Calculation Of Conditional Execution Fee In The _conditionalExecutionFeeHandler Function	High	Fixed	In use
11	Incorrectly Setting Of User TPSLs	High	Fixed	In use
12	Rounding Down Issue Leaving Some Service Charge In APHCore	High	Fixed	In use
13	Rounding Of repayAmount Is Not In Favor Of Lenders	High	Fixed	In use
14	Rounding Of newInterestOwedPerDay Is Not In Favor Of Lenders	High	Fixed	In use
15	Rounding Of collateralSwappedAmountReturn Is Not In Favor Of Protocol	Medium	Fixed	In use
16	Remain The leftOverCollateral As It Is Rounded Down Issue	Medium	Fixed	In use
17	Donation Attack To Increase itpPrice By Claim Rounding Down And Then Multiple Deposit	Medium	Acknowledged	In use
18	The itpBurnAmount Does Not Align With WithdrawAmount	Medium	Fixed	In use

19	Potential Over-Distribution Of Lending Bonuses	Medium	Acknowledged	In use
20	Bypassing The checkStakingAmountSufficient When The PriceFeed Is Paused	Medium	Acknowledged	In use
21	Division By Zero When The Global PriceFeed Is Paused	Medium	Fixed	In use
22	Overpayment For requiredAmount Not Be Refunded	Medium	Fixed	In use
23	Potential Denial Of Liquidate On Blacklisted Loaner	Medium	Acknowledged	In use
24	Avoid Using block.number On Some L2 Chains	Medium	Acknowledged	In use
25	Price Diff Prevention Is Susceptible To Price Manipulation	Medium	Acknowledged	In use
26	TODO Comments Should Be Resolved	Medium	Fixed	In use
27	Insufficient Handling Of User's TPSLs When tpslTimesLimit Changes	Medium	Fixed	In use
28	Trigger Order Fee For Suddenly Open Order Is Kept In Conditional Contract	Medium	Acknowledged	In use
29	Inadequate Handling Of Paused Or Unsupported Price Feeds In _placeOrder Function	Low	Fixed	In use
30	Inconsistent Router Interface Usage In ConditionalFunc And CoreSwappingV3 Contracts	Low	Fixed	In use
31	Recommended Following Best Practices For Upgradeable Smart Contracts	Low	Fixed	In use
32	Unsafe ABI Encoding	Low	Fixed	In use
33	Recommended Event Emissions For Transparency And Traceability	Informational	Fixed	In use
34	Incorrectly Emitted Event Value	Informational	Fixed	In use
35	Recommended Removing Unused Code	Informational	Fixed	In use
36	Inconsistent Usage Of Manager Roles	Informational	Fixed	In use
37	Misspellings And Typos In Codebase	Informational	Fixed	In use
38	Out Of Audit Scope	Informational	Acknowledged	In use

The statuses of the issues are defined as follows:

Fixed: The issue has been completely resolved and has no further complications.

Partially Fixed: The issue has been partially resolved.

Acknowledged: The issue's risk has been reported and acknowledged.

Detailed Result

This section provides all issues that we found in detail.

No. 1	Loss Tracking Precision Mismatch In APHCore		
Risk	Critical	Likelihood	High
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/pool/PoolLendingV2.sol* *contracts/src/core/APHCore.sol*		
Locations	*PoolLendingV2._withdraw L: 352* *APHCore.addLossInUSD L: 98 - 110*		

Detailed Issue

We found that **the implementation of the *addLossInUSD* function of the *APHCore* contract does not support the multiple token precisions.**

To elaborate, The result of the ***lossAmount = (lossAmount * rate) / WEI_UNIT;* will return the *lossAmount* in the precision of itself** as

- ***rate*** is represented by the **18** precisions
- ***WEI_UNIT*** is represented by the **18** precisions
- ***lossAmount*** is represented **according to the *APHPool* token precision**

The precision of the ***lossAmount*** is ***APHPool_token_precision + 18 - 18 = APHPool_token_precision***

Given that the *APHCore* contract can interact with multiple *APHPool* contracts, each of which may involve different precision levels in the amount calculation based on the pool's token precision, we consider the scenarios where the *APHCore* contract interacts with *APHPool* contracts that have varying precision.

As a result, the ***nftsLossInUSD[nftId]*** and ***totalLossInUSD*** values become **inaccurate due to the mixing of the precision amounts of each incoming *lossAmount* from the different *APHPool* contracts.**

PoolLendingV2.sol

```
function _withdraw(

    // (...SNIPPED...)

```

```
    uint256 atpBurnAmount = tokenHolder.pToken > 0
        ? ((withdrawAmount * tokenHolder.atpToken) / (tokenHolder.pToken))
        : 0;
    atpBurnAmount = _burnAtpToken(receiver, nftId, atpBurnAmount, atpPrice);

    uint256 pBurnAmount = _burnPToken(receiver, nftId, withdrawAmount);

    uint256 lossBurnAmount = MathUpgradeable.min(withdrawAmount - actualWithdrawAmount, loss);
    loss -= lossBurnAmount;

    IAPHCore(coreAddress).addLossInUSD(nftId, lossBurnAmount);
```

Listing 1.1 The *addLossInUSD* function calling of the *PoolLendingV2* contract invoking

APHCore.sol

```
function addLossInUSD(uint256 nftId, uint256 lossAmount) external {
    require(poolToAsset[msg.sender] != address(0), "APHCore/caller-is-not-pool");

    uint256 rate;
    {
        (rate, ) = _queryRateUSD(IAPHPool(msg.sender).tokenAddress());
    }
    lossAmount = (lossAmount * rate) / WEI_UNIT;
    nftsLossInUSD[nftId] = nftsLossInUSD[nftId] + lossAmount;
    totalLossInUSD = totalLossInUSD + lossAmount;

    emit AddLossInUSD(address(this), msg.sender, nftId, lossAmount);
}
```

Listing 1.2 The *addLossInUSD* function of the *APHCore* contract

Recommendations

We recommend updating the formula to support the multiple precisions of incoming ***lossAmount*** as shown below.

The formula recommendation:

lossAmount = (lossAmount * rate) / tokenPrecisionUnit[poolToAsset[msg.sender]]; **will return the *lossAmount* in the precision of 18** as

- ***rate*** is represented by the **18** precisions
- ***tokenPrecisionUnit[poolToAsset[msg.sender]]*** is represented **according to the *APHPool* token precision**
- ***poolToAsset[msg.sender]*** returns the address of the *APHPool* caller's underlying/token address.

- ***lossAmount*** is represented **according to the *APHPool* token precision**.

The result precision of the ***lossAmount*** is

APHPool_token_precision + 18 - APHPool_token_precision = 18

APHCore.sol

```
function addLossInUSD(uint256 nftId, uint256 lossAmount) external {
    require(poolToAsset[msg.sender] != address(0),
"APHCore/caller-is-not-pool");

    uint256 rate;
    {
        (rate, ) = _queryRateUSD(IAPHPool(msg.sender).tokenAddress());
    }
    lossAmount = (lossAmount * rate) /
tokenPrecisionUnit[poolToAsset[msg.sender]];
    nftsLossInUSD[nftId] = nftsLossInUSD[nftId] + lossAmount;
    totalLossInUSD = totalLossInUSD + lossAmount;

    emit AddLossInUSD(address(this), msg.sender, nftId, lossAmount);
}
```

Listing 1.3 The improved *addLossInUSD* function of the *APHCore* contract

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *FWX* team adopted our recommendation and fixed this issue.

No. 2	Unable Migration For StakePool To StakePoolV2		
Risk	Critical	Likelihood	High
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/stakepool/StakePool.sol* *contracts/src/stakepool/StakePoolV2.sol*		
Locations	-		

Detailed Issue

The *StakePool2* contract is the new version of the *StakePool* contract. However, we noticed that the *StakePool* is a non-upgradeable contract that can not directly update the new contract logic while still keeping contract states such as the user stakes, and neither *StakePool nor StakePool2* contracts have the migration process.

The issue occurs when the admin invokes the *setNewPool* function of the *Membership* contract (code snippet 2.1) to the *StakePool2* contract without a migration process, **it will force any decision such as staking balance (L192 in the code snippet 2.2) or NFT ranking (code snippet 2.3) that relies on the current pool causes the user may lose benefits from their NFT rank and staking balance that has ever been done from the original pool**.

Membership.sol

```
function setNewPool(address newPool) external onlyAddressTimelockManager
whenNotPaused {
    currentPool = newPool;
    _poolIndex[newPool] = _poolList.length;
    _poolList.push(newPool);

    emit SetNewPool(msg.sender, newPool);
}
```

Listing 2.1 The *setNewPool* function of the *Membership* contract

APHCore.sol

```
function checkStakingAmountSufficient(
    uint256 nftId,
    uint256 newAmount,
    address tokenAddress
) external view returns (uint256) {
    IPriceFeed priceFeed = IPriceFeed(priceFeedAddress);
    {
        (uint256 rate, ) = priceFeed.queryRateUSD(tokenAddress);
        newAmount = (newAmount * rate) / tokenPrecisionUnit[tokenAddress]; // 1e18
    }

    uint256 totalUSDOfUserLent; // 1e18
    for (uint i = 0; i < poolList.length; i++) {
        IAPHPool pool = IAPHPool(poolList[i]);
        uint256 pToken = pool.balancePTokenOf(nftId);
        (uint256 rate, ) = priceFeed.queryRateUSD(pool.tokenAddress());
        totalUSDOfUserLent =
            totalUSDOfUserLent +
            ((pToken * rate) / tokenPrecisionUnit[pool.tokenAddress()]);
    }

    totalUSDOfUserLent += newAmount;
    StakePoolBase.StakeInfo memory stakeInfo = IStakePool(
        IMembership(membershipAddress).currentPool()
    ).getStakeInfo(nftId);

    uint256 currentStakedBalance = stakeInfo.stakeBalance;
    uint256 requireBalance = ((totalUSDOfUserLent * forwStakingMultiplier) / WEI_UNIT);

    require(
        currentStakedBalance >= requireBalance,
        "APHCore/stake-not-matched-minimum-requirement"
    );

    return requireBalance;
}
```

Listing 2.2 The *setNecheckStakingAmountSufficient* function of the *APHCore* contract that uses the current stake balance of the given NFTID for use in decision-making

CoreFutureBaseFunc.sol

```
function _getPositionMargin(
    uint256 nftId,
    bytes32 pairByte,
    bool checkPriceDiff
) internal returns (uint256 margin) {
    GetPositionMarginTmpStruct memory tmp;
    Pair memory pair = pairs[pairByte];
    Position memory pos = positions[nftId][pairByte];
    uint256 COLLATERAL_PRECISION = tokenPrecisionUnit[pair.pair0];
    uint256 UNDERLYING_PRECISION = tokenPrecisionUnit[pair.pair1];
    PositionState memory posState = positionStates[nftId][pos.id];
    require(pos.id != 0 || posState.active,
"CoreTrading/position-is-not-active");

    tmp.wallet = wallets[nftId][pairByte];
    tmp.tradingFee = _getNFTRankInfo(nftId).tradingFee;
    tmp.interestOwed = _calculateFutureInterest(nftId, pairByte);
    (tmp.PNL, tmp.amounts, tmp.rate, tmp.swapFee) = _getUnrealizedPNL(
        nftId,
        pairByte,
        checkPriceDiff
    );
    if (posState.isLong) {
        tmp.feeAmount = tmp.swapFee + ((tmp.amounts[1] * tmp.tradingFee) /
WEI_PERCENT_UNIT);
        tmp.positionSize = (pos.entryPrice * pos.contractSize) /
UNDERLYING_PRECISION;
    } else {
        tmp.feeAmount =
            (pos.borrowAmount * tmp.tradingFee * (pos.entryPrice + tmp.rate)) /
            (UNDERLYING_PRECISION * WEI_PERCENT_UNIT);
        tmp.feeAmount += tmp.swapFee;
        tmp.interestOwed = (tmp.interestOwed * tmp.rate) / UNDERLYING_PRECISION;
        tmp.positionSize = (pos.entryPrice * pos.borrowAmount) /
UNDERLYING_PRECISION;
    }

    (uint256 rate, ) = _queryRateUSD(pair.pair0);
    tmp.feeAmount += (liquidationFee * COLLATERAL_PRECISION) / rate;
    margin = APHLibrary._calculateMargin(
        tmp.wallet,
        pos.collateralSwappedAmount,
        tmp.interestOwed,
        tmp.PNL,
        tmp.positionSize,
        tmp.feeAmount
    );
}
```

Listing 2.3 The *_getPositionMargin* function of the *CoreFutureBaseFunc* contract that decided trading fees from the NFT rank

Recommendations

We recommend the *FWX* team implement the safe migration process that allows users to seamlessly move to the *StakePool2* contract without losing benefits that have ever been done from the original pool.

Reassessment

The FWX team addressed this issue by adding the **migrate** function of **StakePoolV2** contract that allows users to migrate from the **StakePoolV1** to the **StakePoolV2** and explained the migration process as follows:

"*We allow users to migrate staked tokens from StakePoolV1 to StakePoolV2. The migration process begins by settling the staked tokens on both StakePoolV1 and StakePoolV2 separately. Then, we migrate data from StakePoolV1 to StakePoolV2 using four payout patterns for both. If there are any locked tokens in StakePoolV1 during the migration, they will be moved to StakePoolV2 and use the configuration from StakePoolV2 for further settlements.*"

No. 3	Inconsistent Calculation For The Liquidation Fee		
Risk	High	Likelihood	High
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/conditional/ConditionalFunc.sol*		
Locations	*ConditionalFunc._getFreeCollateral L: 307* *ConditionalFunc._getMaxContractSize L: 354 - 355*		

Detailed Issue

We found that the calculation of the liquidation fee in the *_getFreeCollateral* and *_getMaxContractSize* functions of the *ConditionalFunc* (code snippet 3.1) contract is inconsistent with the liquidation fee calculation in the *CoreFutureClosing* (code snippet 3.2) and *CoreFutureBaseFunc* contracts.

To elaborate, the liquidation fee in the *_getFreeCollateral* and *_getMaxContractSize* functions of the *ConditionalFunc* contract is calculated using the following formula:

*LiquidationFee * COLLATERAL_UNIT / **WEI_UNIT***

While in the *CoreFutureClosing* and *CoreFutureBaseFunc* contracts, it is calculated following the formula:

*LiquidationFee * COLLATERAL_UNIT / **COLLATERAL_USD_RATE***

ConditionalFunc.sol

```
339 function _getMaxContractSize(
340     GetMaxContractSizeParams memory params
341 ) internal view returns (uint256 maxContractSize) {
342     IAPHCore aphCore = _getCoreProxy();
343     CoreBase.Pair memory pair = aphCore.pairs(params.pairByte);
344     CoreBase.Position memory pos = aphCore.positions(params.nftId,
params.pairByte);
345     CoreBase.PositionState memory posState =
aphCore.positionStates(params.nftId, pos.id);
346     CoreBase.PositionConfig memory posConfigs =
aphCore.positionConfigs(params.pairByte);
347     IHelperFutureTrade helperFuture =
IHelperFutureTrade(helperFutureTradeAddress);
348     GetMaxContractSizeTmp memory tmp;
349     tmp.tradingFee = _getTradingFee(params.nftId);
```

```
    {
        (tmp.res0, tmp.res1) = _getReserves(1, pair.pair0, pair.pair1);
        tmp.collateralUnit = aphCore.tokenPrecisionUnit(pair.pair0);
        tmp.underlyingUnit = aphCore.tokenPrecisionUnit(pair.pair1);
        tmp.liqFee = aphCore.liquidationFee();
        tmp.liqFee = (tmp.liqFee * tmp.collateralUnit) / WEI_UNIT;
        tmp.actualPrice =
            (tmp.res0 * tmp.underlyingUnit * (WEI_UNIT + (params.slippage) /
100)) /
            (tmp.res1 * WEI_UNIT);

     // (...SNIPPED...)
```

Listing 3.1 The *_getMaxContractSize* function of the *ConditionalFunc* contract

CoreFutureClosing.sol

```
function _liquidatePosition(uint256 nftId, bytes32 pairByte) internal {

    // (...SNIPPED...)

    if (msg.sender != tmp.nftOwner && poolToAsset[msg.sender] == address(0)) {
        // bounty fee
        {
            uint256 wallet = wallets[nftId][pairByte];

            (uint256 rate, ) = _queryRateUSD(tmp.collateralToken);
            uint256 COLLATERAL_PRECISION =
tokenPrecisionUnit[tmp.collateralToken];
            tmp.liquidationFee = (liquidationFee * COLLATERAL_PRECISION) / rate;
```

Listing 3.2 The *_liquidatePosition* function of the *CoreFutureClosing* contract

Recommendations

We recommend updating the liquidation fee calculation to be consistent throughout all logic that uses this value to ensure the results are consistent and accurate.

Reassessment

The *FWX* team adopted our recommendation and fixed this issue.

No. 4	Position Order Opens Out Of Slippage		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Acknowledged
Associated Files	*contracts/src/conditional/ConditionalOrder.sol*		
Locations	*ConditionalOrder._placeOrder L: 76 - 146*		

Detailed Issue

We found that the acceptable price range in the *_placeOrder* function of the *ConditionalOrder* contract does not consider both the ***upperEdge*** and ***lowerEdge*** bounds.

With the condition:

ConditionalOrder._placeOrder **L126: newIsLong ? upperEdge >= rate : lowerEdge <= rate**

consider the following scenarios:

- **LONG Position: If targetPrice = 500, slippage 10%, and the current rate is 440**
  - *upperEdge* = 500 + 10% = 550
  - *lowerEdge* = 500 - 10% = 450
  - as the current rate = 440, the order is incorrectly accepted as ***upperEdge: 550 >= rate: 440***.
- **SHORT Position: If targetPrice = 100, slippage = 10%, and the current rate is 200**
  - *upperEdge* = 100 + 10% = 110
  - *lowerEdge* = 100 - 10% = 90
  - As the current rate is 200, the order is incorrectly accepted as ***lowerEdge: 90 <= rate: 200***.

Consequently, an out-of-bounds price may be used as the entry price of the position, allowing the position to open for the user erroneously.

Moreover, this check is inconsistent with the check to trigger the place limit order in the *_triggerOrder* function of the *ConditionalOrder* contract (code snippet 4.2).

ConditionalOrder.sol

```
function _placeOrder(PlaceOrderStruct memory params) internal {

    // (...SNIPPED...)

        {
        uint256 upperEdge = (params.targetPrice * (WEI_PERCENT_UNIT + params.slipPage)) /
            WEI_PERCENT_UNIT;
        uint256 lowerEdge = (params.targetPrice * (WEI_PERCENT_UNIT - params.slipPage)) /
            WEI_PERCENT_UNIT;

        // open position if the price range is acceptable otherwise place limit order
        if (newIsLong ? upperEdge >= rate : lowerEdge <= rate) {
            _openPositionToPool(nftId, rate, newOrder);
            return;
        } else {
```

Listing 4.1 The *_placeOrder* function of the *ConditionalOrder* contract

ConditionalOrder.sol

```
function _triggerOrder(uint256 nftId, bytes32 pairByte, uint256 index) internal {

    // (...SNIPPED...)

    uint256 rate = _getRateInCollaUnit(pairs.pair1, pairs.pair0);
    if (rate <= targetOrder.upperTargetPrice && rate >= targetOrder.lowerTargetPrice) {
        uint256 beforeOpen = IERC20Upgradeable(pairs.pair0).balanceOf(address(this));
        _openPositionToPool(nftId, rate, targetOrder);
        _deleteOrder(nftId, uint8(index));
        uint256 afterOpen = IERC20Upgradeable(pairs.pair0).balanceOf(address(this));

        _transferOut(msg.sender, pairs.pair0, afterOpen - beforeOpen);
        emit TriggerOrder(msg.sender, targetOrder, uint8(index));
    }
```

Listing 4.2 The *_triggerOrder* function of the *ConditionalOrder* contract

Recommendations

We recommend updating the condition in the *_placeOrder* function to check both the *upperEdge* and *lowerEdge* bounds to ensure that the price is within the acceptable range for both *LONG* and *SHORT* positions

ConditionalOrder.sol

```
function _placeOrder(PlaceOrderStruct memory params) internal {

// (...SNIPPED...)

        if (rate >= lowerEdge && rate <= upperEdge) {
            _openPositionToPool(nftId, rate, newOrder);
            return;
        }

// (...SNIPPED...)

{
```

Listing 4.3 The improved *_placeOrder* function of the *ConditionalOrder* contract

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The FWX team has acknowledged this issue with the statement:

"The design works as intended. We don't set a lower bound for long positions or an upper bound for short positions, as this allows users to receive better rates than expected."

No. 5	Potential Withdraw Collateral Even Reach The Liquidation Through The adjustCollateral Function		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/core/CoreBorrowing.sol*		
Locations	*CoreBorrowing.adjustCollateral L: 132 - 162*		

Detailed Issue

Several borrowing functions of the protocol need to trigger the *_rollover* function of the *CoreBorrowing* contract (code snippet 5.1) before performing when overdue date to update the *loan.interestOwed* (L587 and 606 in the code snippet 5.1) which is used to calculate the LTV, especially, the liquidation criteria uses the LTV as part of the decision.

The *adjustCollateral* function of the *CoreBorrowing* contract is one of the borrowing functions that allows users to adjust their collateral as long as the loan LTV after removal remains lower than *maxLTV* from *LoanConfig*.

However, we found that the internal *_adjustCollateral* function of the *CoreBorrowing* contract (code snippet 5.2) did not invoke the *_rollover* function to update the loan information, consequently, the *_isLoanLTVExceedTargetLTV* function (L541 - 551 in the code snippet 5.2) may use the outdated loan information to decide.

To understand this issue, consider the following attack scenario:

1. Alice borrowed the WETH tokens from the WETH pool

Assume the loan is overdue and reaches the liquidation criteria while no one rolls over for this loan to update loan information. From this, Alice's loan should be liquidated, and can not withdraw their collateral.

2. Instead of repaying their loan, Alice calls the *adjustCollateral* function to withdraw their collateral and bypass the *_isLoanLTVExceedTargetLTV* validation. **Consequently, this may cause lenders or liquidators to lose their benefits from the outdated loan information.**

CoreBorrowing.sol

```
function _rollover(
    uint256 loanId,
    uint256 nftId,
    address caller
) internal returns (uint256 delayInterest, uint256 bountyReward) {
    Loan storage loan = loans[nftId][loanId];
    require(loanExts[nftId][loanId].active == true, "CoreBorrowing/loan-is-closed");
    address bountyRewardTokenAddress;

    LoanConfig storage loanConfig = loanConfigs[loan.borrowTokenAddress][
        loan.collateralTokenAddress
    ];

    // This loan is overdue, the penalty is charged to loan's owner.
    if (block.timestamp > loan.rolloverTimestamp) {
        delayInterest = ((block.timestamp - loan.rolloverTimestamp) * loan.owedPerDay) / 1 days;
        bountyReward = (delayInterest * loanConfig.penaltyFeeRate) / WEI_PERCENT_UNIT;

        if (caller == _getTokenOwnership(nftId) || poolToAsset[caller] != address(0)) {
            // Caller is owner, collect delay interest to interestOwed
            loan.interestOwed += delayInterest + bountyReward;
            bountyRewardTokenAddress = loan.borrowTokenAddress;

            // Set bountyReward to zero since no bountyReward is for liquidator.
            bountyReward = 0;
        } else {
            (uint256 rate, ) = _queryRate(loan.collateralTokenAddress, loan.borrowTokenAddress);
            // check dex rate with oracle
            _validatePriceDiff(rate, loan, loan.borrowAmount);
            // Caller is liquidator, bounty fee is sent to liquidator in form of collateral token equal to bountyFee
            bountyReward = IPriceFeed(priceFeedAddress).queryReturn(
                loan.borrowTokenAddress,
                loan.collateralTokenAddress,
                bountyReward
            );

            totalCollateralHold[loan.collateralTokenAddress] -= bountyReward;

            loan.interestOwed += delayInterest;
            if (bountyReward > loan.collateralAmount) {
                bountyReward = loan.collateralAmount;
            }
            loan.collateralAmount -= bountyReward;
```

```
            bountyRewardTokenAddress = loan.collateralTokenAddress;
        }
    }
    address poolAddress = assetToPool[loan.borrowTokenAddress];
    PoolStat storage poolStat = poolStats[poolAddress];
    poolStat.updatedTimestamp = block.timestamp;

    // Calculate new interest owed per day to this loan
    (uint256 interestRate, ) = IAPHPool(poolAddress).calculateInterest(0);
    uint256 interestOwedPerDay = (loan.borrowAmount * interestRate) / (WEI_PERCENT_UNIT * 365);

    loan.rolloverTimestamp = uint64(block.timestamp + loanDuration);

    poolStat.borrowInterestOwedPerDay =
        poolStat.borrowInterestOwedPerDay -
        loan.owedPerDay +
        interestOwedPerDay;

    loan.owedPerDay = interestOwedPerDay;
    loan.lastSettleTimestamp = uint64(block.timestamp);

    emit Rollover(
        _getTokenOwnership(nftId),
        nftId,
        loanId,
        caller,
        loan.borrowTokenAddress,
        delayInterest,
        bountyReward,
        bountyRewardTokenAddress,
        interestOwedPerDay
    );
}
```

Listing 5.1 The *_rollover* function of the *CoreBorrowing* contract that is used to update loan information

CoreBorrowing.sol

```
function _adjustCollateral(
    uint256 loanId,
    uint256 nftId,
    uint256 collateralAdjustAmount,
    bool isAdd
) internal returns (Loan memory) {
    Loan storage loan = loans[nftId][loanId];
    require(loanExts[nftId][loanId].active == true, "CoreBorrowing/loan-is-closed");

```

```
    _settleBorrowInterest(loan);

    LoanConfig storage loanConfig = loanConfigs[loan.borrowTokenAddress][
        loan.collateralTokenAddress
    ];

    if (isAdd) {
        loan.collateralAmount += collateralAdjustAmount;

        totalCollateralHold[loan.collateralTokenAddress] +=
collateralAdjustAmount;
    } else {
        loan.collateralAmount -= collateralAdjustAmount;
        require(
            loan.collateralAmount >=
                _getMinimumCollateralInCollateralUnit(
                    loan.collateralTokenAddress,
                    loan.borrowTokenAddress
                ),
            "CoreBorrowing/collateral-less-than-minimum-collateral"
        );

        totalCollateralHold[loan.collateralTokenAddress] -=
collateralAdjustAmount;

        (uint256 rate, uint256 precision) = _queryRate(
            loan.collateralTokenAddress,
            loan.borrowTokenAddress
        );
        // check dex rate with oracle
        _validatePriceDiff(rate, loan, loan.borrowAmount);

        require(
            _isLoanLTVExceedTargetLTV(
                loan.borrowAmount,
                loan.collateralAmount,
                MathUpgradeable.max(loan.interestOwed, loan.minInterest),
                loanConfig.maxLTV +

IStakePool(IMembership(membershipAddress).currentPool()).getMaxLTVBonus(
                        nftId
                    ),
                rate,
                precision
            ) == false,
            "CoreBorrowing/loan-LTV-is-exceed-maxLTV"
        );
    }

    emit AdjustCollateral(
        msg.sender,
```

```
        nftId,
        loanId,
        isAdd,
        loan.collateralTokenAddress,
        collateralAdjustAmount
    );
    return loan;
}
```

Listing 5.2 The *_adjustCollateral* function of the *CoreBorrowing* contract

Recommendations

We recommend invoking the *_rollover* function (L540 - 542 in the code snippet below) once before validating the LTV with the *_isLoanLTVExceedTargetLTV* function to ensure the use of up-to-date loan information to decide.

CoreBorrowing.sol

```
501 function _adjustCollateral(
502     uint256 loanId,
503     uint256 nftId,
504     uint256 collateralAdjustAmount,
505     bool isAdd
506 ) internal returns (Loan memory) {
507     Loan storage loan = loans[nftId][loanId];
508     require(loanExts[nftId][loanId].active == true,
"CoreBorrowing/loan-is-closed");
509
510     _settleBorrowInterest(loan);
511
512     LoanConfig storage loanConfig = loanConfigs[loan.borrowTokenAddress][
513         loan.collateralTokenAddress
514     ];
515
516     if (isAdd) {
517         loan.collateralAmount += collateralAdjustAmount;
518
519         totalCollateralHold[loan.collateralTokenAddress] +=
collateralAdjustAmount;
520     } else {

        // (...SNIPPED...)

540         if (loan.rolloverTimestamp < block.timestamp) {
541             _rollover(loanId, nftId, msg.sender);
542         }
543
544         require(
```

```
                _isLoanLTVExceedTargetLTV(
                    loan.borrowAmount,
                    loan.collateralAmount,
                    MathUpgradeable.max(loan.interestOwed, loan.minInterest),
                    loanConfig.maxLTV +
IStakePool(IMembership(membershipAddress).currentPool()).getMaxLTVBonus(
                            nftId
                        ),
                    rate,
                    precision
                ) == false,
                "CoreBorrowing/loan-LTV-is-exceed-maxLTV"
            );
        }

        emit AdjustCollateral(
            msg.sender,
            nftId,
            loanId,
            isAdd,
            loan.collateralTokenAddress,
            collateralAdjustAmount
        );
        return loan;
}
```

Listing 5.3 The improved *_adjustCollateral* function of the *CoreBorrowing* contract

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *FWX* team updates the *_adjustCollateral* function to fix this issue as shown in the code snippet below.

CoreBorrowing.sol

```
function _adjustCollateral(
    uint256 loanId,
    uint256 nftId,
    uint256 collateralAdjustAmount,
    bool isAdd
) internal returns (Loan memory) {

    // (...SNIPPED...)
```

```
        uint256 delayInterest;
        if (block.timestamp > loan.rolloverTimestamp) {
            delayInterest =
                ((block.timestamp - loan.rolloverTimestamp) * loan.owedPerDay) /
                1 days;
        }

        require(
            _isLoanLTVExceedTargetLTV(
                loan.borrowAmount,
                loan.collateralAmount,
                MathUpgradeable.max(loan.interestOwed + delayInterest,
loan.minInterest),
                loanConfig.maxLTV +

IStakePool(IMembership(membershipAddress).currentPool()).getMaxLTVBonus(
                        nftId
                    ),
                rate,
                precision
            ) == false,
            "CoreBorrowing/loan-LTV-is-exceed-maxLTV"
        );

       // (...SNIPPED...)
}
```

Listing 5.4 The fixed _*adjustCollateral* function of the *CoreBorrowing* contract

No. 6	Misclassification Of Forwarder As Liquidator In Repay And Rollover Functions		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Acknowledged
Associated Files	*contracts/src/core/CoreBorrowing.sol*		
Locations	*CoreBorrowing._repay L: 413* *CoreBorrowing._rollover L: 585*		

Detailed Issue

The *Forwarder* prepares the *repay* and *rollover* functions, enabling users to repay and rollover their loans via the *Forwarder*.

However, when the user calls the *repay* or *rollover* functions through the *Forwarder* and the loan is overdue, the *_rollover* function is triggered (L413 in code snippet 6.2). During this process, the caller is checked with the condition

if (caller == _getTokenOwnership(nftId) || poolToAsset[caller] != address(0)) (L585 in code snippet 6.3).

The caller parameter received from the *_repay* and/or *rollover* function will be the *Forwarder*, which means the condition will be *false*, and the caller will be treated as a liquidator in the else clause.

As a result, the *Forwarder* will be incorrectly recognized as a liquidator. This misclassification can lead to financial discrepancies and potentially unfair penalties applied to the user's loan.

CoreBorrowing.sol

```
function repay(
        uint256 loanId,
        uint256 nftId,
        uint256 repayAmount,
        bool isOnlyInterest
    )
        external
        payable
        whenFuncNotPaused(msg.sig)
        nonReentrant
        returns (uint256 borrowPaid, uint256 interestPaid)
    {
        // ! call by forwarder(gasless)
```

```
        if (!forwarderAddressWhitelist[msg.sender]) {
            nftId = _getUsableToken(msg.sender, nftId);
        }
        Loan storage loan = loans[nftId][loanId];
        require(
            loan.borrowTokenAddress == wethAddress || msg.value == 0,
            "CoreBorrowing/no-support-transfering-ether-in"
        );

        bool isLoanClosed;
        uint256 tmpCollateralAmount = loan.collateralAmount;
        (borrowPaid, interestPaid, isLoanClosed) = _repay(
            loanId,
            nftId,
            repayAmount,
            isOnlyInterest
        );

    // (...SNIPPED...)

}
```

Listing 6.1 The *repay* function of the *CoreBorrowing* contract

CoreBorrowing.sol

```
function _repay(
    uint256 loanId,
    uint256 nftId,
    uint256 repayAmount,
    bool isOnlyInterest
) internal returns (uint256 borrowPaid, uint256 interestPaid, bool isLoanClosed)
{
    Loan storage loan = loans[nftId][loanId];
    PoolStat storage poolStat = poolStats[assetToPool[loan.borrowTokenAddress]];
    poolStat.updatedTimestamp = block.timestamp;

    require(loanExts[nftId][loanId].active == true,
"CoreBorrowing/loan-is-closed");

    _settleBorrowInterest(loan);

        // Rollover loan if it is overdue.
    if (loan.rolloverTimestamp < block.timestamp) {
        _rollover(loanId, nftId, msg.sender);
    }

    // (...SNIPPED...)
}
```

Listing 6.2 The *_repay* function of the *CoreBorrowing* contract

CoreBorrowing.sol

```
function _rollover(
        uint256 loanId,
        uint256 nftId,
        address caller
    ) internal returns (uint256 delayInterest, uint256 bountyReward) {
        Loan storage loan = loans[nftId][loanId];
        require(loanExts[nftId][loanId].active == true, "CoreBorrowing/loan-is-closed");
        address bountyRewardTokenAddress;

        LoanConfig storage loanConfig = loanConfigs[loan.borrowTokenAddress][
            loan.collateralTokenAddress
        ];

        // This loan is overdue, the penalty is charged to loan's owner.
        if (block.timestamp > loan.rolloverTimestamp) {
            delayInterest = ((block.timestamp - loan.rolloverTimestamp) * loan.owedPerDay) / 1 days;
            bountyReward = (delayInterest * loanConfig.penaltyFeeRate) / WEI_PERCENT_UNIT;

            if (caller == _getTokenOwnership(nftId) || poolToAsset[caller] != address(0)) {
                // Caller is owner, collect delay interest to interestOwed
                loan.interestOwed += delayInterest + bountyReward;
                bountyRewardTokenAddress = loan.borrowTokenAddress;

                // Set bountyReward to zero since no bountyReward is for liquidator.
                bountyReward = 0;
            } else {
                (uint256 rate, ) = _queryRate(loan.collateralTokenAddress, loan.borrowTokenAddress);
                // check dex rate with oracle
                _validatePriceDiff(rate, loan, loan.borrowAmount);
                // Caller is liquidator, bounty fee is sent to liquidator in form of collateral token equal to bountyFee
                bountyReward = IPriceFeed(priceFeedAddress).queryReturn(
                    loan.borrowTokenAddress,
                    loan.collateralTokenAddress,
                    bountyReward
                );

                totalCollateralHold[loan.collateralTokenAddress] -= bountyReward;

                loan.interestOwed += delayInterest;
                if (bountyReward > loan.collateralAmount) {
                    bountyReward = loan.collateralAmount;
                }
```

```
                loan.collateralAmount -= bountyReward;
                bountyRewardTokenAddress = loan.collateralTokenAddress;
            }
        }
    // (...SNIPPED...)
}
```

Listing 6.3 The *_rollover* function of the *CoreBorrowing* contract

Recommendations

For the recommendation, we cannot suggest specific code changes because it may cause further issues. We recommend adding a mechanism to correctly identify the original user behind the *Forwarder* call.

Moreover, the team has the flexibility to adopt and adapt these recommendations based on their specific business requirements.

Reassessment

The *FWX* team has acknowledged this issue with the statement as *“For caller == forwarder, rollover in normal case, since forwarder doesn’t have liquidate function”*.

No. 7	Incorrect Bounty Fee Distribution In Liquidation Scenarios		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Acknowledged
Associated Files	*contracts/src/core/CoreFutureClosing.sol* *contracts/src/gasless/ForwarderCore.sol* *contracts/src/conditional/ConditionalTPSL.sol*		
Locations	*CoreFutureClosing._liquidatePosition L: 120 - 255* *ForwarderCore.closePosition L: 151* *ConditionalTPSL._trigger L: 83*		

Detailed Issue

The *CoreFutureClosing* contract provides the *liquidatePosition* function, allowing anyone to call and liquidate a position if it is in a liquidation state (L14 in code snippet 7.1).

This function differentiates the caller into two groups:

1. **Position Owner (NFT Owner)**
2. **Non-Position Owner (Liquidator)**: The liquidator can be an EOA account or a smart contract (L176 in code 7.1).

In this liquidation process, a bounty fee is provided to the liquidator as an incentive (L204 in code snippet 7.1).

However, we identified issues with the bounty fee when this function is called from scenarios involving the *Forwarder* and *ConditionalTPSL* contracts. Here are the details:

1. **Caller: Forwarder**
   Scenario: The *Forwarder* calls the *closePosition* function and the current state of the position is in liquidation, the *Forwarder* becomes the liquidator, and the bounty fee for the liquidator is transferred to the *Forwarder*.
   Result: The bounty fee becomes stuck in the *Forwarder* contract because the *Forwarder* cannot withdraw funds.

2. **Caller: Forwarder -> ConditionalTPSL**
   Scenario: The *Forwarder* calls *setTPSL* through the *ConditionalTPSL* contract to set take profit/stop loss. If the position is in a liquidation state when triggered, the bounty fee is transferred to the

Forwarder.
Result: The fund becomes stuck in the *Forwarder* contract due to the lack of withdrawal capability.

3. **Caller: NFT Owner (Position Owner) -> ConditionalTPSL**
Scenario: The NFT Owner calls *setTPSL* through the *ConditionalTPSL* contract to set take profit/stop loss. If the position is in a liquidation state when triggered, the bounty fee is transferred to the NFT Owner.
Result: Normally, the liquidation process does not provide a bounty fee to the NFT Owner, but this scenario does, leading to an inconsistency and incorrect distribution of fees.

In summary, the current implementation of the *_liquidatePosition* function in *CoreFutureClosing* contract can lead to mismanagement of bounty fees when called by different entities like the *Forwarder* and *ConditionalTPSL*. These scenarios result in bounty fees becoming stuck or incorrectly distributed.

CoreFutureClosing.sol

```
function liquidatePosition(uint256 nftId, bytes32 pairByte) external {
    _liquidatePosition(nftId, pairByte);
}

// (...SNIPPED...)

function _liquidatePosition(uint256 nftId, bytes32 pairByte) internal {
       Position storage pos = positions[nftId][pairByte];
       PositionState storage posState = positionStates[nftId][pos.id];

       _verifyPosition(pos.id, posState.active);

       require(
           _getPositionMargin(nftId, pairByte, false) <
               positionConfigs[pairByte].maintenanceMargin,
           "CoreTrading/current-margin-too-high"
       );

       _settleFutureTradeInterest(nftId, pairByte);

       LiquidatePositionTmpStruct memory tmp;
       APHLibrary.ClosePositionResponse memory result;

       // ! check is Short or Long
       // close position
       tmp.posId = pos.id;
       tmp.nftOwner = _getTokenOwnership(nftId);
       {
           tmp.closingSize = posState.isLong ? pos.contractSize :
pos.borrowAmount;
           APHLibrary.ClosePositionParams memory params =
```

```
APHLibrary.ClosePositionParams(
                nftId,
                pairByte,
                tmp.posId,
                tmp.closingSize,
                _getNFTRankInfo(nftId).tradingFee,
                true
            );

            result = posState.isLong ? _closeLong(params) : _closeShort(params);

            emit ClosePosition(
                tmp.nftOwner,
                nftId,
                tmp.posId,
                params.closingSize,
                result.rate,
                result.PNL,
                posState.isLong,
                false,
                posState.pairByte,
                result.collateralSwappedAmountReturn,
                result.router
            );
        }

        tmp.collateralToken = pairs[posState.pairByte].pair0;
        tmp.underlyingToken = pairs[posState.pairByte].pair1;

        // collect a liquidation fee when the liquidator is not the position's owner. (The position's owner is not calling any functions to make a self-liquidation.)
        // conditions:
        //  - "msg.sender != tmp.nftOwner": calling `closePosition` and `liquidatePosition` directly.
        //  - "poolToAsset[msg.sender] == address(0)": calling `openPosition` on the opposite side to the `APHPool`, then the APHPool calls `closePosition` to `APHCore`.
        if (msg.sender != tmp.nftOwner && poolToAsset[msg.sender] == address(0)) {
            // bounty fee
            {
                uint256 wallet = wallets[nftId][pairByte];

                (uint256 rate, ) = _queryRateUSD(tmp.collateralToken);
                uint256 COLLATERAL_PRECISION = tokenPrecisionUnit[tmp.collateralToken];
                tmp.liquidationFee = (liquidationFee * COLLATERAL_PRECISION) / rate;

                if (tmp.liquidationFee >= wallet) {
```

```
                    tmp.liquidationFee = wallet;
                    wallet = 0;
                } else {
                    wallet = wallet - tmp.liquidationFee;

                    tmp.bountyFeeToProtocol =
                        (wallet * positionConfigs[pairByte].bountyFeeRateToProtocol) /
                        WEI_PERCENT_UNIT;
                    tmp.bountyFeeToLiquidator =
                        (wallet * positionConfigs[pairByte].bountyFeeRateToLiquidator) /
                        WEI_PERCENT_UNIT;

                    wallet = wallet - tmp.bountyFeeToProtocol - tmp.bountyFeeToLiquidator;
                }

                tmp.bountyFeeToLiquidator += tmp.liquidationFee;
                _updateWallet(nftId, pairByte, wallet);
                if (tmp.bountyFeeToLiquidator > 0) {
                    _safeTransfer(tmp.collateralToken, msg.sender, tmp.bountyFeeToLiquidator);
                }

                if (tmp.bountyFeeToProtocol > 0) {
                    _safeTransfer(tmp.collateralToken, feeVaultAddress, tmp.bountyFeeToProtocol);
                    IFeeVault(feeVaultAddress).settleFeeProfitAndFeeAuction(
                        tmp.collateralToken,
                        tmp.bountyFeeToProtocol,
                        0
                    );
                }
            }
        }

        {
            address transferToken = posState.isLong ? tmp.collateralToken : tmp.underlyingToken;
            // settle interest and transfer
            _settleAndTransferFutureTradeFee(
                transferToken,
                result.feeToIntVault,
                result.feeToProfitVault
            );

            // transfer repay amount
            _safeTransfer(transferToken, assetToPool[transferToken], result.repayAmount);
        }
```

```
        emit LiquidatePosition(
            tmp.nftOwner,
            nftId,
            tmp.posId,
            posState.isLong,
            msg.sender,
            tmp.closingSize,
            result.rate,
            posState.pairByte,
            result.router
        );

        emit CollectFees(
            tmp.nftOwner,
            nftId,
            pos.id,
            posState.pairByte,
            uint128(result.tradingFee),
            uint128(result.swapFee),
            uint128(result.interestPaid),
            uint128(tmp.liquidationFee),
            uint128(tmp.bountyFeeToProtocol),
            uint128(tmp.bountyFeeToLiquidator)
        );
}
```

Listing 7.1 The *_liquidatePosition* function of the *CoreFutureClosing* contract

Recommendations

For the recommendation, we cannot suggest specific code changes because it may cause further issues. We recommend implementing mechanisms to properly handle the scenarios where the *Forwarder* and *ConditionalTPSL* are involved in the liquidation process to ensure the correct distribution and accessibility of the bounty fee.

Moreover, the team has the flexibility to adopt and adapt these recommendations based on their specific business requirements.

Reassessment

The *FWX* team has acknowledged this issue with the statement as *"For caller == forwarder, transfer fee to nft owner, since forwarder don't have liquidate function"*.

No. 8	Service Charge Lockup In Forwarder Contract Due To setTPSL Function		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/conditional/ConditionalTPSL.sol* *contracts/src/gasless/ForwarderConditionalTrading.sol*		
Locations	*ConditionalTPSL.setTPSL L: 9 - 19* *ConditionalTPSL._setTPSL L: 100 - 122* *ConditionalTPSL._trigger L: 41 - 98* *ForwarderConditionalTrading.setTPSL L: 9 - 21*		

Detailed Issue

The *setTPSL* function in the *ConditionalTPSL* contract allows setting values that can trigger within the same transaction if the TPSLs are within the specified range of conditions (L9 in code snippet 8.2). In this process, the system transfers the service charge fee back to the caller of the *setTPSL* function.

However, we found an issue in the following scenario:

- A user already has a long position with an entry price of 400 TokenA/TokenB.
- The *Forwarder* prepares the *setTPSL* function that allows the user to set TPSLs via the *Forwarder* (L9 in code snippet 8.1).
- **The user uses a *Forwarder* to call the *setTPSL* function in the *ConditionalTPSL* contract**, setting the target price for take profit at 450 TokenA/TokenB.
- During the call to the *setTPSL* function, the current price of TokenA/TokenB is 450, which means this transaction will automatically trigger the TPSLs to close the current position and take profit.

From the scenario above, **if a user sets their TPSLs via the *Forwarder,* the caller will be a *Forwarder* and the service charge from the triggered TPSLs will be transferred to the *Forwarder (L86 in code snippet 8.2)***. The *Forwarder* does not have a function to withdraw the service charge, which means the service charge will become stuck in the *Forwarder* contract.

Note: TPSLs is Take Profit/Stop Loss

ForwarderConditionalTrading.sol

```
contract ForwarderConditionalTrading is ForwarderFunc {
    function setTPSL(
        address smartWalletAddress,
        address paidExecutionFeeTokenAddress,
        uint256 nftId,
        bytes32 pairByte,
        ConditionalBase.TPSL[] memory tpsls
    ) external onylExecutionToken(paidExecutionFeeTokenAddress) onlyNFTOwner(nftId) nonReentrant {
        uint256 gasUsed = gasleft();
        IConditional ct = IConditional(conditionalAddress);
        ct.setTPSL(nftId, pairByte, tpsls);
        gasUsed = gasUsed - gasleft();
        _chargeExecutionFee(smartWalletAddress, paidExecutionFeeTokenAddress, gasUsed);
}
```

Listing 8.1 The *setTPSL* function of the *ForwarderConditionalTrading* contract

ConditionalTPSL.sol

```
contract ConditionalTPSL is ConditionalFunc {
   // ! EXTERNAL FUNCTION
   function setTPSL(
       uint256 nftID,
       bytes32 pairByte,
       TPSL[] memory _tpsl
   ) external nonReentrant whenFuncNotPaused(msg.sig) {
       IAPHCore aphCore = _getCoreProxy();
       if (!aphCore.forwarderAddressWhitelist(msg.sender)) {
           nftID = _getUsableToken(msg.sender, nftID);
       }
       _setTPSL(nftID, pairByte, _tpsl);
   }

// (...SNIPPED...)

function _setTPSL(uint256 nftID, bytes32 pairByte, TPSL[] memory _tpsl) internal
{
       require(_tpsl.length <= tpslTimesLimit, "ConditionalTrading/out-of-limit-tpsl-times");

       // nftID = _getUsableToken(msg.sender, nftID);

       TPSL[] storage TPSLthisPos = TPSLs[nftID][pairByte];
       TPSL[] memory oldTPSL = TPSLthisPos;
       _assignTPSLMemoryToStorage(nftID, pairByte, _tpsl);
```

```

        emit SetTPSL(msg.sender, nftID, pairByte, oldTPSL, _tpsl);

        _trigger(nftID, pairByte);
    }
```

Listing 8.2 The *setTPSL and _setTPSL* functions of the *ConditionalTPSL* contract

ConditionalTPSL.sol

```
function _trigger(uint256 nftID, bytes32 pairByte) internal {
    IAPHCore aphCore = _getCoreProxy();
    CoreBase.Position memory pos = aphCore.positions(nftID, pairByte);
    CoreBase.PositionState memory posState = aphCore.positionStates(nftID,
pos.id);

        if (posState.active == false) {
            _clearTPSL(nftID, pairByte);
            return;
        }
        TPSL[] storage tpsl = TPSLs[nftID][pairByte];
        require(tpsl.length != 0, "ConditionalTrading/no-tpsl-exists");

        uint256 rate = _getRateInCollaUnit(
            posState.isLong ? pos.swapTokenAddress : pos.borrowTokenAddress,
            pos.collateralTokenAddress
        );

        uint256 closingSize;

        TPSL[] memory previousTPSL = TPSLs[nftID][pairByte];

        for (uint8 index = 0; index < tpsl.length; index++) {
            if (tpsl[index].targetPrice > 0 && tpsl[index].contractSize > 0) {
                if (
                    _isTriggerable(posState.isLong, tpsl[index].isTP, rate,
tpsl[index].targetPrice)
                ) {
                    closingSize += tpsl[index].contractSize;
                    tpsl[index] = TPSL(0, 0, false, 0);
                }
            }
        }

        if (closingSize > 0) {
            // Call close position
            CoreBase.Pair memory pairs = aphCore.pairs(pairByte);
            uint256 beforeOpen =
IERC20Upgradeable(pairs.pair0).balanceOf(address(this));
```

```
            bytes memory data = abi.encodeWithSignature(
                "closePosition(uint256,uint256,uint256)",
                nftID,
                pos.id,
                closingSize
            );
            _call(coreAddress, data);

            uint256 afterOpen =
IERC20Upgradeable(pairs.pair0).balanceOf(address(this));
            _transferOut(msg.sender, pairs.pair0, afterOpen - beforeOpen);

            pos = aphCore.positions(nftID, pairByte);
            if (pos.contractSize <= 1) {
                tpsl = TPSLs[0][0];
            } else {
                _assignTPSLMemoryToStorage(nftID, pairByte, tpsl);
            }

            emit SetTPSL(msg.sender, nftID, pairByte, previousTPSL, tpsl);
        }
        return;
    }
```

Listing 8.3 The *_trigger* function of the *ConditionalTPSL* contract

Recommendations

For the recommendation, we cannot suggest specific code changes because it may cause further issues. We recommend implementing a function within the *Forwarder* contract to allow the withdrawal of service charges. Additionally, consider adding logic to the *ConditionalTPSL* contract to prevent the transfer of service charges to *Forwarders* without withdrawal functionality.

Moreover, the team has the flexibility to adopt and adapt these recommendations based on their specific business requirements.

Reassessment

The *FWX* team fixed this issue by adding the *withdrawToken* function, which enables the admin to withdraw service charges through the *Timelock* mechanism as shown in the code snippet below.

Forwarder.sol

```
function withdrawToken(
    address tokenAddress,
    uint256 amount,
    address recipient
) external onlyAddressTimelockManager {
    amount = Math.min(IERC20(tokenAddress).balanceOf(address(this)), amount);
    if (tokenAddress == wethAddress) {
        IWethERC20(wethAddress).safeTransfer(wethHandlerAddress, amount);
        IWethHandler(payable(wethHandlerAddress)).withdrawETH(recipient,
amount);
    } else {
        IERC20(tokenAddress).safeTransfer(recipient, amount);
    }
}
```

Listing 8.4 The *withdrawToken* function of the *Forwarder* contract

No. 9	Incompatibility Issues In PoolLendingV2		
Risk	High	Likelihood	High
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/pool/PoolLendingV2.sol*		
Locations	*Several functions in PoolLendingV2 contract*		

Detailed Issue

We found that the *activateRank* function in the *PoolLendingV2* contract is not compatible with the *APHPoolProxy* contract. The incompatibility of the returned data results in the inability to interact with the *activateRank* function from the *PoolLendingV2* contract.

Additionally the *PoolLendingV2* contract does not support calls from the *Forwarder*. As a result, calls from the *Forwarder* to *PoolLendingV2* may work incorrectly or unexpectedly. The functions listed below should support *Forwarder* calls, similar to the implementation in *PoolLending* version 1:

1. The *activateRank* function
2. The *deposit* function
3. The *withdraw* function
4. The *claimAllInterest* function
5. The *claimTokenInterest* function
6. The *claimForwInterest* function

These crucial functions should be supported for *Forwarder* calls to ensure proper functionality.

APHPoolProxy.sol

```
function activateRank(uint256 nftId) external returns (WithdrawResult memory
result) {
    bytes memory data = abi.encodeCall(PoolLending.activateRank, nftId);

    data = _delegateCall(poolLendingAddress, data);
    result = abi.decode(data, (WithdrawResult));
}
```

Listing 9.1 The *activateRank* function of the *APHPoolProxy* contract

PoolLending.sol

```
function activateRank(
    uint256 nftId
)
    external
    nonReentrant
    whenFuncNotPaused(msg.sig)
    settleForwInterest
    returns (WithdrawResult memory result)
{
    // ! call by forwarder(gasless)
    if (!IAPHCore(coreAddress).forwarderAddressWhitelist(msg.sender)) {
        nftId = _getUsableToken(msg.sender, nftId);
    }
    // (...SNIPPED...)
}
```

Listing 9.2 The *activateRank* function of the *PoolLending* contract that supports *Forwarder*

PoolLendingV2.sol

```
function activateRank(
    uint256 nftId
) external nonReentrant whenFuncNotPaused(msg.sig) settleForwInterest returns (uint8 newRank) {
    nftId = _getUsableToken(msg.sender, nftId);
    WithdrawResult memory result;
    (result, newRank) = _activateRank(msg.sender, nftId);

    // (...SNIPPED...)
}
```

Listing 9.3 The *activateRank* function of the *PoolLendingV2* contract that does not support *Forwarder*

Recommendations

We recommend updating the *activateRank* function in the *PoolLendingV2* contract to be fully compatible with the *APHPoolProxy* contract by aligning the returned data structures and formats.

Additionally, we recommend supporting *Forwarder* calls for the specified functions. This will ensure that the functionality matches the expected behavior and aligns with the implementation in *PoolLending* version 1.

Moreover, the team has the flexibility to adopt and adapt these recommendations based on their specific business requirements.

Reassessment

The *FWX* team adopted our recommendation and fixed this issue.

No. 10	Incorrect Calculation Of Conditional Execution Fee In The _conditionalExecutionFeeHandler Function		
Risk	High	Likelihood	High
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/core/CoreFutureBaseFunc.sol*		
Locations	*CoreFutureBaseFunc._conditionalExecutionFeeHandler L: 245 - 264*		

Detailed Issue

We found that the *_conditionalExecutionFeeHandler* function calculates the conditional execution fee using the value of the ***WETH* token in *USD***, normalizes the unit to the collateral unit (L254 - 255 in code snippet 10.1), and then subtracts this value from the collateral of the given wallet (L258 in code snippet 10.1). However, we identified an inconsistency in this calculation process.

The **wallet of NFT ID for each *pairBytes* contains only the collateral token**, meaning the unit for each wallet is based on the collateral unit of each pairBytes.

However, the deduction of the conditional execution fee from the wallet in the *_conditionalExecutionFeeHandler* function is incorrect as **it subtracts the *USD* value (*serviceCharge* in USD) from the collateral value (wallet contains collateral token)**.

This leads to an inaccurate fee deduction as the *serviceCharge* retrieves the *WETH* value in terms of *USD* value, while the wallet holds only the collateral value. Consequently, pools with high-value tokens as collateral will incur higher fee values and vice versa.

Consider the following scenario

Assume the following:

- *WEI_UNIT* = 1e18
- *WETH_PRECISION* = 1e18
- *USD_RATE_PRECISION* = 1e18
- 1 *WETH* = 3000 *USD* = 300 * 1e18
- ***serviceCharge*** = 0.001 WETH = **1e15**
- Collateral token = *BNB* with 18 precision

1. The rate is normalized to the collateral precision:

   *rate = (rate * tokenPrecisionUnit[collateralToken]) / tokenPrecisionUnit[wethAddress];*

   *rate = (3000 * 1e18 (usd)) * 1e18 / 1e18 = **3000 * 1e18 USD in Collateral Precision***

2. Applying the proportion of ***serviceCharge***:

   *serviceCharge = (rate * serviceCharge) / WEI_UNIT;*

   *serviceCharge = [**3000 * 1e18]** * 1e15 / 1e18 = **3 * 1e18** => **3 USD in Collateral Precision***

3. The serviceCharge is incorrectly deducted from the collateral wallet as:

   *_updateWallet(nftId, pairByte, **wallet - serviceCharge**);*

   wallet: BNB value (Collateral) - serviceCharge: USD value in Collateral Precision

CoreFutureBaseFunc.sol

```
function _conditionalExecutionFeeHandler(
    address caller,
    address collateralToken,
    uint256 serviceCharge,
    uint256 nftId,
    bytes32 pairByte
) internal {
    if (serviceCharge > 0) {
        (uint256 rate, ) = _queryRateUSD(wethAddress);
        rate = (rate * tokenPrecisionUnit[collateralToken]) /
tokenPrecisionUnit[wethAddress];
        serviceCharge = (rate * serviceCharge) / WEI_UNIT;

        uint256 wallet = wallets[nftId][pairByte];
        _updateWallet(nftId, pairByte, wallet - serviceCharge);

        serviceCharge = serviceCharge / 2;
        _settleAndTransferFutureTradeFee(collateralToken, 0, serviceCharge);
        _transferOut(caller, collateralToken, serviceCharge);
    }
}
```

Listing 10.1 The *_conditionalExecutionFeeHandlerfunction* of the *CoreFutureBaseFunc* contract

Recommendations

We recommend converting the *serviceCharge* value that is used in subtraction from each wallet to contain the same value in terms of collateral token's value to accurately deduct the conditional execution fee from the collateral wallet, preventing overcharging or undercharging based on collateral value.

Moreover, the team has the flexibility to adopt and adapt these recommendations based on their specific business requirements.

Reassessment

The *FWX* team adopted our recommendation and fixed this issue.

No. 11	Incorrectly Setting Of User TPSLs		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/conditional/ConditionalTPSL.sol* *contracts/src/conditional/ConditionalFunc.sol*		
Locations	*ConditionalTPSL._setTPSL L: 107* *ConditionalTPSL._trigger L: 92* *ConditionalFunc._assignTPSLMemoryToStorage L: 160 - 173*		

Detailed Issue

We found an incorrect value assignment of user *TPSLs* when they set their *TPSLs* for the first time as ***TPSLthisPos*** is declared as a storage pointed to the ***TPSLs[ ][ ]*** (L165 in the code snippet 11.1).

However, this storage point will change to point to the **initial *TPSLs* (*TPSLs[0][0]*)** (L167 in the code snippet 11.1). As a result, the assignment process at L170 in the code snippet 11.1 modifies the state of the **initial *TPSLs* (*TPSLs[0][0]*)** instead of the user's *TPSLs* and then assigns this modified state back to the user's *TPSLs* (L172 in the code snippet 11.1)

Consider the following scenario:

1. The initial *TPSLs: TPSLs[0][0]* contains the ***n*** *TPSLs* with the default values:
    a. *TPSLs[0][0] = [TPSL_i = 0(..default_value..), .., TPSL_n-1(..default_value..)]*

2. User A sets their *TPSLs* by providing a new *_tpsl* array of full limit length (***n*** length array)
    a. Since User A has never set their *TPSLs*, the condition at L166 will be executed and the storage pointer will point to the *TPSLs[0][0]*.
    b. The loop at L169 - 171 assigns Use A's given values to the *TPSLs[0][0]*.
    c. User A's *TPSLs* are reassigned to the values pointing to the modified *TPSLs[0][0]* (modified in step **b.**)(L172)

3. User B sets their *TPSLs* by providing new *_tpsl* array with fewer than ***n*** limit length (***n-2*** length array)
    a. Since User B has never set their *TPSLs*, the condition at L166 will be executed and the storage pointer will point to the *TPSLs[0][0]*. (already modified in step 2.)
    b. The loop at L169 - 171 assigns Use B given values to the *TPSLs[0][0],* **but only loop through the *n-2* array length** so the *TPSLs[0][0]* is only modified at ***index [0:n-3]***, leaving the ***index [n-2, n-1]*** with the value of previous modification.

c. User B's *TPSLs* are reassigned values pointing to the *TPSLs[0][0]*, containing both User B's *TPSLs* and the rest of the *TPSLs[0][0]* that were previously assigned by User A.

4. The unexpected assigned values of the User B's TPSLs can be triggered through the *_trigger* function

In conclusion, users can retrieve unexpected *TPSL* settings from previous modifications made by other users, and the initial *TPSLs* ***(TPSLs[0][0])*** will be modified each time a new user sets their *TPSLs* for the first time. This leads to the initial values being dynamically changed by users.

ConditionalFunc.sol

```
function _assignTPSLMemoryToStorage(
    uint256 nftID,
    bytes32 pairByte,
    TPSL[] memory _tpsl
) internal {
    TPSL[] storage TPSLthisPos = TPSLs[nftID][pairByte];
    if (TPSLthisPos.length == 0) {
        TPSLthisPos = TPSLs[0][0];
    }
    for (uint i = 0; i < _tpsl.length; i++) {
        TPSLthisPos[i] = _tpsl[i];
    }
    TPSLs[nftID][pairByte] = TPSLthisPos;
}
```

Listing 11.1 The *_assignTPSLMemoryToStorage* of the *ConditionalFunc* contract

ConditionalTPSL.sol

```
function _setTPSL(uint256 nftID, bytes32 pairByte, TPSL[] memory _tpsl) internal {
    require(_tpsl.length <= tpslTimesLimit, "ConditionalTrading/out-of-limit-tpsl-times");

    // nftID = _getUsableToken(msg.sender, nftID);

    TPSL[] storage TPSLthisPos = TPSLs[nftID][pairByte];
    TPSL[] memory oldTPSL = TPSLthisPos;
    _assignTPSLMemoryToStorage(nftID, pairByte, _tpsl);

    emit SetTPSL(msg.sender, nftID, pairByte, oldTPSL, _tpsl);

    _trigger(nftID, pairByte);
}
```

Listing 11.2 The *_setTPSL* of the *ConditionalTPSL* contract

ConditionalTPSL.sol

```
function _trigger(uint256 nftID, bytes32 pairByte) internal {
    IAPHCore aphCore = _getCoreProxy();
    CoreBase.Position memory pos = aphCore.positions(nftID, pairByte);
    CoreBase.PositionState memory posState = aphCore.positionStates(nftID, pos.id);

    if (posState.active == false) {
        _clearTPSL(nftID, pairByte);
        return;
    }
    TPSL[] storage tpsl = TPSLs[nftID][pairByte];
    require(tpsl.length != 0, "ConditionalTrading/no-tpsl-exists");
```

Listing 11.3 The *_trigger* of the *ConditionalTPSL* contract

Recommendations

We recommend revisiting the logic for setting user TPSLs to ensure that each user's TPSLs are stored and modified independently. Additionally, we recommend revising the process to prevent modifications to the ***initial TPSLs***, retaining them as the default values.

Reassessment

The *FWX* team fixed this issue by revising the *_assignTPSLMemoryToStorage* function to correctly update the TPSLs.

No. 12	Rounding Down Issue Leaving Some Service Charge In APHCore		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/core/CoreFutureBaseFunc.sol*		
Locations	*CoreFutureBaseFunc._conditionalExecutionFeeHandler L: 260 - 262*		

Detailed Issue

As illustrated in code snippet 12.1, the **service charge for executing functions in the *Conditional* contract is evenly split between the executor and the fee vault, each receiving 50%. However**, **due to the rounding down** behavior of the Solidity language, **a remainder of tokens can accumulate in the *Conditional* contract**.

For instance, consider the following scenario:
in Line 248, a service charge of 5 tokens is computed
in Line 258, deducts 5 tokens from the position owner's wallet
in Line 260, the calculation of the service charge divided by two (5 / 2 = 2.5) results in 2.5, which is rounded down to 2.

Consequently, 2 tokens are transferred to both the fee vault and the executor, leaving 1 token remaining in the *Conditional* contract

CoreFutureBaseFunc.sol

```
245 function _conditionalExecutionFeeHandler(
246     address caller,
247     address collateralToken,
248     uint256 serviceCharge,
249     uint256 nftId,
250     bytes32 pairByte
251 ) internal {
252     if (serviceCharge > 0) {
253         (uint256 rate, ) = _queryRateUSD(wethAddress);
254         rate = (rate * tokenPrecisionUnit[collateralToken]) /
    tokenPrecisionUnit[wethAddress];
255         serviceCharge = (rate * serviceCharge) / WEI_UNIT;
256 
257         uint256 wallet = wallets[nftId][pairByte];
```

```
        _updateWallet(nftId, pairByte, wallet - serviceCharge);

        serviceCharge = serviceCharge / 2;
        _settleAndTransferFutureTradeFee(collateralToken, 0, serviceCharge);
        _transferOut(caller, collateralToken, serviceCharge);
    }
}
```

Listing 12.1 The *_conditionalExecutionFeeHandler* function of the *CoreFutureBaseFunc* contract

Recommendations

The division of the service charge between the fee vault and the executor in the *Conditional* contract involves an inevitable rounding error.

The decision on who benefits from the leftover tokens belongs to the FWX team. The FWX team can either allocate the remaining tokens to a specific party or keep track of the accumulated tokens in the *Conditional* contract for later distribution.

Nonetheless, **the FWX team has the flexibility to adopt and adapt these recommendations based on their specific business requirements and priorities**.

Reassessment

The *FWX* team fixed this issue by calculating the *feeToProtocol* (L258 in the code snippet 12.2) for transfer to the fee vault, with the remaining *serviceCharge* (L260 in the code snippet 12.2) allocated to the executor as shown in the code snippet below.

CoreFutureBaseFunc.sol

```
244 function _conditionalExecutionFeeHandler(
245     address caller,
246     address collateralToken,
247     uint256 serviceCharge,
248     uint256 nftId,
249     bytes32 pairByte
250 ) internal {
251     if (serviceCharge > 0) {
252         (uint256 rate, ) = _queryRateUSD(collateralToken);
253         serviceCharge = (serviceCharge * tokenPrecisionUnit[collateralToken]) /
rate;
254
255         uint256 wallet = wallets[nftId][pairByte];
256         _updateWallet(nftId, pairByte, wallet - serviceCharge);
257
```

```
        uint256 feeToProtocol = serviceCharge - (serviceCharge / 2);
        // serviceCharge will be half of previous service charge
        serviceCharge = serviceCharge - feeToProtocol;

        _settleAndTransferFutureTradeFee(collateralToken, 0, feeToProtocol);
        _transferOut(caller, collateralToken, serviceCharge);
  }
}
```

Listing 12.2 The improved *_conditionalExecutionFeeHandler* function of the *CoreFutureBaseFunc* contract

No. 13	Rounding Of repayAmount Is Not In Favor Of Lenders		
Risk	High	Likelihood	High
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/core/CoreFutureClosing.sol*		
Locations	*CoreFutureClosing._closeLong L: 302*		

Detailed Issue

In the *CoreFutureClosing* contract, when a position owner closes their long position, the contract calculates the *repayAmount*, which indicates how much principal needs to be repaid to lenders. Lenders' tokens are used in positions to provide leverage. Code snippet 13.1 L302 illustrates this *repayAmount* calculation.

However, we discovered that **the calculation of *repayAmount* for closing long positions does not round the number in favor of the lenders**. This flaw could allow **position owners to close portions of the position and profit without fully repaying the borrowed tokens to lenders**.

For instance, consider the following scenario:

1. *pos.contractSize* is 500e18 while *pos.borrowAmount* is 100e18 and *params.closingSize* is 1.
2. *result.repayAmount = (params.closingSize * pos.borrowAmount) / pos.contractSize;*
   result.repayAmount = (1 * 100e18) /500e18;
   ***result.repayAmount* = 0;**
3. *actualCollateral -= result.repayAmount;*
   ***actualCollateral* -= 0; the position owner does not have to pay the principal for closing a portion of the position.**
4. Repeat.

There are practical considerations that make this attack challenging to execute on the live network, including:

1. In the final close position call that reduces *pos.contractSize* to zero, the position owner must repay the entire *pos.borrowAmount*. To avoid this, the position owner could choose not to fully close the position.
2. The gas cost might be so high, making the attack unprofitable if the attacker pays a normal gas price. However, if the attacker is a miner then their gas price is zero, so this attack costs them nothing in gas.

CoreFutureClosing.sol

```
function _closeLong(
    APHLibrary.ClosePositionParams memory params
) internal returns (APHLibrary.ClosePositionResponse memory result) {
    // (...SNIPPET...)
    result.repayAmount = (params.closingSize * pos.borrowAmount) / pos.contractSize;

    // calculate real actualCollateral
    actualCollateral = actualCollateral + amounts[1] - result.swapFee;
    bool isCritical = actualCollateral < result.repayAmount;

    if (isCritical == false) {
        actualCollateral -= result.repayAmount;
        (actualCollateral, result.tradingFee) = _cascadeActualCollateral(
            pos,
            posState,
            actualCollateral,
            result.tradingFee
        );

        _updateWallet(params.nftId, params.pairByte, actualCollateral);

        uint256 newInterestOwedPerDay = (pos.interestOwePerDay *
            (pos.contractSize - params.closingSize)) / pos.contractSize;
        uint256 collateralSwappedAmountReturn = MathUpgradeable.min(
            (pos.collateralSwappedAmount * params.closingSize) / pos.contractSize,
            pos.collateralSwappedAmount
        );

        // update pool stat
        poolStat.totalBorrowAmountFromTrading -= result.repayAmount;
        poolStat.borrowInterestOwedPerDayFromTrading -= (pos.interestOwePerDay -
            newInterestOwedPerDay);

        // update position
        pos.borrowAmount -= result.repayAmount;
        pos.collateralSwappedAmount -= collateralSwappedAmountReturn;
        pos.interestOwePerDay = newInterestOwedPerDay;
        pos.contractSize -= params.closingSize;
```

Listing 13.1 The *_closeLong* function of the *CoreFutureClosing* contract

Recommendations

The calculation of the *repayAmount* involves an inevitable rounding error. The decision on who benefits from the leftover tokens belongs to the *FWX* team.

We recommend adjusting the code such that the calculated ***repayAmount* is always rounded up instead of being truncated as shown in code snippet 13.2 L303 - 305.**

However, this recommendation will consume more gas to close positions. So, the *FWX* team has the flexibility to adopt and adapt these recommendations based on their specific business requirements and priorities.

CoreFutureClosing.sol

```
257 function _closeLong(
258     APHLibrary.ClosePositionParams memory params
259 ) internal returns (APHLibrary.ClosePositionResponse memory result) {
        // (...SNIPPET...)
302     result.repayAmount = (params.closingSize * pos.borrowAmount) /
    pos.contractSize;
303     if ((params.closingSize * pos.borrowAmount) % pos.contractSize != 0) {
304         result.repayAmount += 1;
305     }
306     // calculate real actualCollateral
307     actualCollateral = actualCollateral + amounts[1] - result.swapFee;
308     bool isCritical = actualCollateral < result.repayAmount;
309 
310     if (isCritical == false) {
311         actualCollateral -= result.repayAmount;
312         (actualCollateral, result.tradingFee) = _cascadeActualCollateral(
313             pos,
314             posState,
315             actualCollateral,
316             result.tradingFee
317         );
318 
319         _updateWallet(params.nftId, params.pairByte, actualCollateral);
320 
321         uint256 newInterestOwedPerDay = (pos.interestOwePerDay *
                (pos.contractSize - params.closingSize)) / pos.contractSize;
322         uint256 collateralSwappedAmountReturn = MathUpgradeable.min(
323             (pos.collateralSwappedAmount * params.closingSize) /
324 pos.contractSize,
325             pos.collateralSwappedAmount
326         );
327 
328         // update pool stat
329         poolStat.totalBorrowAmountFromTrading -= result.repayAmount;
330         poolStat.borrowInterestOwedPerDayFromTrading -= (pos.interestOwePerDay -
```

```
                newInterestOwedPerDay);

        // update position
        pos.borrowAmount -= result.repayAmount;
        pos.collateralSwappedAmount -= collateralSwappedAmountReturn;
        pos.interestOwePerDay = newInterestOwedPerDay;
        pos.contractSize -= params.closingSize;
```

Listing 13.2 The recommended fix in _*closeLong* function of the *CoreFutureClosing* contract

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *FWX* team adopted our recommendation and fixed this issue.

No. 14	Rounding Of newInterestOwedPerDay Is Not In Favor Of Lenders		
Risk	High	Likelihood	High
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/core/CoreFutureClosing.sol*		
Locations	*CoreFutureClosing._closeLong L: 319* *CoreFutureClosing._closeShort L: 526*		

Detailed Issue

In the *CoreFutureClosing* contract, when a position owner closes their long/short position, the contract calculates *newInterestOwedPerDay* to find the interest per day after closing the position. This calculation is illustrated in the following code snippet 14.1 L319 and code snippet 14.2 L528 - 530.

However, **we discovered that the calculation of *newInterestOwedPerDay* does not round the number in favor of the lenders. This flaw benefits the position owner by paying less interest.**

For instance, consider this scenario based on code snippet 14.1:

1. pos.interestOwePerDay = 333, pos.contractSize = 500e18 and params.closingSize = 100e18
2. uint256 newInterestOwedPerDay = (pos.interestOwePerDay * (pos.contractSize - params.closingSize)) / pos.contractSize;
   uint256 newInterestOwedPerDay = (333 * (500e18 - 100e18)) / 500e18;
   uint256 newInterestOwedPerDay = 266.4 which is rounded down to 266.

This truncated interest per day will cost the position owner to pay less interest. This rounding is in favor of position owners instead of lenders. Note that a similar scenario could be applied to *_closeShort* in code snippet 14.2.

CoreFutureClosing.sol

```
257 function _closeLong(
258     APHLibrary.ClosePositionParams memory params
259 ) internal returns (APHLibrary.ClosePositionResponse memory result) {
        // (...SNIPPET...)
302     result.repayAmount = (params.closingSize * pos.borrowAmount) /
    pos.contractSize;
```

```

    // calculate real actualCollateral
    actualCollateral = actualCollateral + amounts[1] - result.swapFee;
    bool isCritical = actualCollateral < result.repayAmount;

    if (isCritical == false) {
        actualCollateral -= result.repayAmount;
        (actualCollateral, result.tradingFee) = _cascadeActualCollateral(
            pos,
            posState,
            actualCollateral,
            result.tradingFee
        );

        _updateWallet(params.nftId, params.pairByte, actualCollateral);

        uint256 newInterestOwedPerDay = (pos.interestOwePerDay *
            (pos.contractSize - params.closingSize)) / pos.contractSize;
        uint256 collateralSwappedAmountReturn = MathUpgradeable.min(
            (pos.collateralSwappedAmount * params.closingSize) /
pos.contractSize,
            pos.collateralSwappedAmount
        );

        // update pool stat
        poolStat.totalBorrowAmountFromTrading -= result.repayAmount;
        poolStat.borrowInterestOwedPerDayFromTrading -= (pos.interestOwePerDay -
            newInterestOwedPerDay);

        // update position
        pos.borrowAmount -= result.repayAmount;
        pos.collateralSwappedAmount -= collateralSwappedAmountReturn;
        pos.interestOwePerDay = newInterestOwedPerDay;
        pos.contractSize -= params.closingSize;
```

Listing 14.1 The *_closeLong* function of the *CoreFutureClosing* contract

CoreFutureClosing.sol

```
function _closeShort(
    APHLibrary.ClosePositionParams memory params
) internal returns (APHLibrary.ClosePositionResponse memory result) {
        // (...SNIPPET...)
        // update pool stat
        tmp.newInterestOwedPerDay =
            (pos.interestOwePerDay * (pos.borrowAmount - result.repayAmount)) /
            pos.borrowAmount;
        poolStat.totalBorrowAmountFromTrading -= result.repayAmount;
        poolStat.totalInterestPaidFromTrading += (result.feeToIntVault);
```

```
        poolStat.borrowInterestOwedPerDayFromTrading -= (pos.interestOwePerDay -
            tmp.newInterestOwedPerDay);

        // update position
        pos.contractSize -= tmp.closingSize;
        pos.borrowAmount -= result.repayAmount;
        pos.interestOwePerDay = tmp.newInterestOwedPerDay;
```

Listing 14.2 The *_closeShort* function of the *CoreFutureClosing* contract

Recommendations

The calculation of the *newInterestOwedPerDay* involves an inevitable rounding error. The decision on who benefits from the leftover tokens belongs to the FWX team.

We recommend adjusting the code such that the calculated ***newInterestOwedPerDay*** **is always rounded up instead of being truncated. Furthermore, to save more gas we could avoid checking for fractional values and round up every time. This is illustrated in code snippet 14.3 L320 and code snippet 14.4 L528.**

However, this extra rounding will consume more gas to close positions. So, the FWX team has the flexibility to adopt these recommendations based on their specific business requirements and priorities.

CoreFutureClosing.sol

```
257 function _closeLong(
258     APHLibrary.ClosePositionParams memory params
259 ) internal returns (APHLibrary.ClosePositionResponse memory result) {
        // (...SNIPPET...)
302     result.repayAmount = (params.closingSize * pos.borrowAmount) /
pos.contractSize;
303
304     // calculate real actualCollateral
305     actualCollateral = actualCollateral + amounts[1] - result.swapFee;
306     bool isCritical = actualCollateral < result.repayAmount;
307
308     if (isCritical == false) {
309         actualCollateral -= result.repayAmount;
310         (actualCollateral, result.tradingFee) = _cascadeActualCollateral(
311             pos,
312             posState,
313             actualCollateral,
314             result.tradingFee
315         );
316
317         _updateWallet(params.nftId, params.pairByte, actualCollateral);
```

```
        uint256 newInterestOwedPerDay = ((pos.interestOwePerDay *
            (pos.contractSize - params.closingSize)) / pos.contractSize) + 1;
        uint256 collateralSwappedAmountReturn = MathUpgradeable.min(
            (pos.collateralSwappedAmount * params.closingSize) /
pos.contractSize,
            pos.collateralSwappedAmount
        );

        // update pool stat
        poolStat.totalBorrowAmountFromTrading -= result.repayAmount;
        poolStat.borrowInterestOwedPerDayFromTrading -= (pos.interestOwePerDay -
            newInterestOwedPerDay);

        // update position
        pos.borrowAmount -= result.repayAmount;
        pos.collateralSwappedAmount -= collateralSwappedAmountReturn;
        pos.interestOwePerDay = newInterestOwedPerDay;
        pos.contractSize -= params.closingSize;
```

Listing 14.3 The recommended fix in *_closeLong* function of the *CoreFutureClosing* contract

CoreFutureClosing.sol

```
function _closeShort(
    APHLibrary.ClosePositionParams memory params
) internal returns (APHLibrary.ClosePositionResponse memory result) {
        // (...SNIPPET...)
        // update pool stat
        tmp.newInterestOwedPerDay =
            ((pos.interestOwePerDay * (pos.borrowAmount - result.repayAmount)) /
            pos.borrowAmount) + 1;
        poolStat.totalBorrowAmountFromTrading -= result.repayAmount;
        poolStat.totalInterestPaidFromTrading += (result.feeToIntVault);
        poolStat.borrowInterestOwedPerDayFromTrading -= (pos.interestOwePerDay -
            tmp.newInterestOwedPerDay);

        // update position
        pos.contractSize -= tmp.closingSize;
        pos.borrowAmount -= result.repayAmount;
        pos.interestOwePerDay = tmp.newInterestOwedPerDay;
```

Listing 14.4 The recommended fix in *_closeShort* function of the *CoreFutureClosing* contract

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *FWX* team adopted our recommendation and fixed this issue.

No. 15	Rounding Of collateralSwappedAmountReturn Is Not In Favor Of Protocol		
Risk	Medium	Likelihood	High
		Impact	Low
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/core/CoreFutureClosing.sol*		
Locations	*CoreFutureClosing._closeLong L: 321 - 324* *CoreFutureClosing._closeShort L: 520 - 524*		

Detailed Issue

In the *CoreFutureClosing* contract, when a position owner closes their long/short position, the contract calculates *collateralSwappedAmountReturn* to find the interest per day after closing the position. This calculation is illustrated in the following code snippets (code snippet 15.1 L321-324 and code snippet 15.2 L520-524).

However, **we discovered that the calculation of *collateralSwappedAmountReturn* does not round the number in favor of the protocol. This flaw benefits the position owner by having a higher margin than expected.**

For instance, consider this scenario based on code snippet 15.1 L322

1. pos.collateralSwappedAmount = 4.4e18, params.closingSize = 333 and pos.contractSize = 22e18
2. **(pos.collateralSwappedAmount * params.closingSize) / pos.contractSize = ?**
   = 4.4e18 * 333 / 22e18
   = **666.6 which is rounded down to 666**
3. uint256 collateralSwappedAmountReturn = MathUpgradeable.min(666, pos.collateralSwappedAmount)
   uint256 collateralSwappedAmountReturn = MathUpgradeable.min(666,4.4e18)
   uint256 collateralSwappedAmountReturn = 666
4. pos.collateralSwappedAmount -= collateralSwappedAmountReturn;
   pos.collateralSwappedAmount -= 666;

This truncated *collateralSwappedAmountReturn* will make position owners have more *pos.collateralSwappedAmount* than expected. This results in position owners having higher position margins than expected. Because *pos.collateralSwappedAmount* is used in calculating position margin (code snippet 15.3 L207).

This rounding is in favor of position owners instead of the protocol. Note that a similar scenario could be applied to *_closeShort* in code snippet 15.2.

CoreFutureClosing.sol

```
function _closeLong(
    APHLibrary.ClosePositionParams memory params
) internal returns (APHLibrary.ClosePositionResponse memory result) {
    // (...SNIPPET...)
    result.repayAmount = (params.closingSize * pos.borrowAmount) / pos.contractSize;

    // calculate real actualCollateral
    actualCollateral = actualCollateral + amounts[1] - result.swapFee;
    bool isCritical = actualCollateral < result.repayAmount;

    if (isCritical == false) {
        actualCollateral -= result.repayAmount;
        (actualCollateral, result.tradingFee) = _cascadeActualCollateral(
            pos,
            posState,
            actualCollateral,
            result.tradingFee
        );

        _updateWallet(params.nftId, params.pairByte, actualCollateral);

        uint256 newInterestOwedPerDay = (pos.interestOwePerDay *
            (pos.contractSize - params.closingSize)) / pos.contractSize;
        uint256 collateralSwappedAmountReturn = MathUpgradeable.min(
            (pos.collateralSwappedAmount * params.closingSize) / pos.contractSize,
            pos.collateralSwappedAmount
        );

        // update pool stat
        poolStat.totalBorrowAmountFromTrading -= result.repayAmount;
        poolStat.borrowInterestOwedPerDayFromTrading -= (pos.interestOwePerDay -
            newInterestOwedPerDay);

        // update position
        pos.borrowAmount -= result.repayAmount;
        pos.collateralSwappedAmount -= collateralSwappedAmountReturn;
        pos.interestOwePerDay = newInterestOwedPerDay;
        pos.contractSize -= params.closingSize;
```

Listing 15.1 The *_closeLong* function of the *CoreFutureClosing* contract

CoreFutureClosing.sol

```
function _closeShort(
    APHLibrary.ClosePositionParams memory params
) internal returns (APHLibrary.ClosePositionResponse memory result) {
            // (...SNIPPET...)
            result.collateralSwappedAmountReturn = MathUpgradeable.min(
                (pos.collateralSwappedAmount * result.repayAmount) /
pos.borrowAmount,
                pos.collateralSwappedAmount
            );
            // (...SNIPPET...)
            pos.collateralSwappedAmount -= result.collateralSwappedAmountReturn;
```

Listing 15.2 The *_closeShort* function of the *CoreFutureClosing* contract

CoreFutureBaseFunc.sol

```
function _getPositionMargin(
    uint256 nftId,
    bytes32 pairByte,
    bool checkPriceDiff
) internal returns (uint256 margin) {
    // (...SNIPPET...)
    margin = APHLibrary._calculateMargin(
        tmp.wallet,
        pos.collateralSwappedAmount,
        tmp.interestOwed,
        tmp.PNL,
        tmp.positionSize,
        tmp.feeAmount
    );
```

Listing 15.3 The *_getPositionMargin* function of the *CoreFutureBaseFunc* contract.

Recommendations

The calculation of the *collateralSwappedAmountReturn* involves an inevitable rounding error. The decision on who benefits from the leftover tokens belongs to the FWX team.

We recommend adjusting the code such that the calculated ***collateralSwappedAmountReturn* is rounded up instead of being truncated as shown in code snippet 15.4 L322 and 15.5 L521.**

However, this extra rounding will consume more gas to close positions. So, the FWX team has the flexibility to adopt and adapt these recommendations based on their specific business requirements and priorities.

CoreFutureClosing.sol

```
function _closeLong(
    APHLibrary.ClosePositionParams memory params
) internal returns (APHLibrary.ClosePositionResponse memory result) {
    // (...SNIPPET...)
    result.repayAmount = (params.closingSize * pos.borrowAmount) / pos.contractSize;

    // calculate real actualCollateral
    actualCollateral = actualCollateral + amounts[1] - result.swapFee;
    bool isCritical = actualCollateral < result.repayAmount;

    if (isCritical == false) {
        actualCollateral -= result.repayAmount;
        (actualCollateral, result.tradingFee) = _cascadeActualCollateral(
            pos,
            posState,
            actualCollateral,
            result.tradingFee
        );

        _updateWallet(params.nftId, params.pairByte, actualCollateral);

        uint256 newInterestOwedPerDay = (pos.interestOwePerDay *
            (pos.contractSize - params.closingSize)) / pos.contractSize;
        uint256 collateralSwappedAmountReturn = MathUpgradeable.min(
            ((pos.collateralSwappedAmount * params.closingSize) / pos.contractSize) + 1,
            pos.collateralSwappedAmount
        );

        // update pool stat
        poolStat.totalBorrowAmountFromTrading -= result.repayAmount;
        poolStat.borrowInterestOwedPerDayFromTrading -= (pos.interestOwePerDay -
            newInterestOwedPerDay);

        // update position
        pos.borrowAmount -= result.repayAmount;
        pos.collateralSwappedAmount -= collateralSwappedAmountReturn;
        pos.interestOwePerDay = newInterestOwedPerDay;
        pos.contractSize -= params.closingSize;
```

Listing 15.4 The recommended fix in *_closeLong* function of the *CoreFutureClosing* contract

CoreFutureClosing.sol

```
function _closeShort(
    APHLibrary.ClosePositionParams memory params
) internal returns (APHLibrary.ClosePositionResponse memory result) {
            // (...SNIPPET...)
            result.collateralSwappedAmountReturn = MathUpgradeable.min(
                ((pos.collateralSwappedAmount * result.repayAmount) /
pos.borrowAmount) + 1,
                pos.collateralSwappedAmount
            );
            // (...SNIPPET...)
            pos.collateralSwappedAmount -= result.collateralSwappedAmountReturn;
```

Listing 15.5 The recommended fix in _*closeShort* function of the *CoreFutureClosing* contract

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *FWX* team adopted our recommendation and fixed this issue.

No. 16	Remain The leftOverCollateral As It Is Rounded Down Issue		
Risk	Medium	Likelihood	Medium
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/core/CoreBorrowing.sol*		
Locations	*CoreBorrowing._liquidate L: 201 - 244* *CoreBorrowing._liquidate L: 644 - 807*		

Detailed Issue

In the *CoreBorrowing* contract, when a position is liquidated, the contract calculates *bountyReward*, *bountyToProtocol*, and *leftOverCollateral* to determine the rewards for the liquidator, the protocol, and the remaining collateral for the position owner, respectively.

However, we discovered that the calculation of *bountyReward*, *bountyToProtocol* and *leftOverCollateral* involves rounding down, which leaves a small amount of tokens in the contract.

For instance, consider a scenario where

1. Line 223 of code snippet 16.1 is called to calculate *bountyReward*, *bountyToProtocol* and *leftOverCollateral*.
2. Line 796 of code snippet 16.2 is called to calculate the *bountyReward*.
3. Let's assume that:
   leftOverCollateral = 99
   loanConfig.bountyFeeRateToLiquidator = 10e18
   leftOverCollateral as 99 may look too low compared to the config for the live network contract. However, we use a low number to better visualize this scenario. This rounding error will also affect similar scenario with high number
4. bountyReward += (99 * 10e18) / 100e18
   bountyReward += 9.9 which is rounded down to 9
   bountyReward += 9
5. Line 799 of code snippet 16.2 is called to calculate the bountyToProtocol.
6. Let's assume that loanConfig.bountyFeeRateToProtocol = 3e18.
7. bountyToProtocol += (99 * 3e18) / 100e18
   bountyToProtocol += 2.97 which is rounded down to 2
   bountyToProtocol += 2

8. Line 802 of code snippet 16.2 is called to calculate the leftOverCollateral.
9. leftOverCollateral -= ((10e18 + 3e18) * 99) / 100e18

   leftOverCollateral -= 12.87 which is rounded down to 12

   leftOverCollateral -= 12
10. Line 220 - 222 of code snippet 16.1 will use these calculated values to send funds. As leftOverCollateral value is initially 99, it will be splitted as:

    send 9 tokens as bountyReward to liquidator (Line 234 code snippet 16.1).

    send 2 tokens as bountyToProtocol to the protocol (Line 236 code snippet 16.1).

    leftOverCollateral -= 12, 99 - 12 = 87, so **send 87 tokens to the position owner** (L243 in code snippet 16.1).

In this scenario, 99 tokens are split into 9, 2, and 87 tokens for different recipients. **However, there will be 1 token left in the APHCore contract because 99 - (9 + 2 + 87) = 1.**

CoreBorrowing.sol

```
201 function liquidate(
202     uint256 loanId,
203     uint256 nftId
204 )
205     external
206     whenFuncNotPaused(msg.sig)
207     nonReentrant
208     returns (
209         uint256 repayBorrow,
210         uint256 repayInterest,
211         uint256 bountyReward,
212         uint256 bountyToProtocol,
213         uint256 leftOverCollateral
214     )
215 {
216     Loan storage loan = loans[nftId][loanId];
217     (
218         repayBorrow,
219         repayInterest,
220         bountyReward,
221         bountyToProtocol,
222         leftOverCollateral
223     ) = _liquidate(loanId, nftId);
224 
225     IERC20Upgradeable(loan.borrowTokenAddress).safeTransfer(
226         assetToPool[loan.borrowTokenAddress],
227         repayBorrow
228     );
229     IERC20Upgradeable(loan.borrowTokenAddress).safeTransfer(
230         _getInterestVault(assetToPool[loan.borrowTokenAddress]),
231         repayInterest
232     );
233 
```

```
    _transferOut(msg.sender, loan.collateralTokenAddress, bountyReward);
    if (bountyToProtocol > 0) {
        _safeTransfer(loan.collateralTokenAddress, feeVaultAddress, bountyToProtocol);
        IFeeVault(feeVaultAddress).settleFeeProfitAndFeeAuction(
            loan.collateralTokenAddress,
            bountyToProtocol,
            0
        );
    }
    _transferOut(_getTokenOwnership(nftId), loan.collateralTokenAddress, leftOverCollateral);
}
```

Listing 16.1 The *liquidate* function of the *CoreFutureClosing* contract

CoreBorrowing.sol

```
function _liquidate(
    uint256 loanId,
    uint256 nftId
)
    internal
    returns (
        uint256 repayBorrow,
        uint256 repayInterest,
        uint256 bountyReward,
        uint256 bountyToProtocol,
        uint256 leftOverCollateral
    )
{
        // (...SNIPPET...)
        if (loanLiquidationFee >= leftOverCollateral) {
        // (...SNIPPET...)
        } else {
            bountyReward += loanLiquidationFee;
            leftOverCollateral -= loanLiquidationFee;
            bountyReward +=
                (leftOverCollateral * loanConfig.bountyFeeRateToLiquidator) /
                WEI_PERCENT_UNIT;
            bountyToProtocol +=
                (leftOverCollateral * loanConfig.bountyFeeRateToProtocol) /
                WEI_PERCENT_UNIT;
            leftOverCollateral -=
                ((loanConfig.bountyFeeRateToLiquidator +
                    loanConfig.bountyFeeRateToProtocol) * leftOverCollateral) /
                WEI_PERCENT_UNIT;
        }
```

Listing 16.2 The *_liquidate* function of the *CoreBorrowing* contract

Recommendations

The calculation of the *bountyReward*, *bountyToProtocol* and *leftOverCollateral* involves an inevitable rounding error. **We recommend that leftovers should be prioritized to the protocol and liquidator over position owner.**

However, the decision on who benefits from the leftover tokens belongs to the FWX team. We recommend the FWX to adopt and adapt these recommendations based on their specific business requirements and priorities.

Reassessment

The *FWX* team adopted our recommendation and fixed this issue.

No. 17	Donation Attack To Increase itpPrice By Claim Rounding Down And Then Multiple Deposit		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Acknowledged
Associated Files	*contracts/src/pool/PoolLending.sol* *contracts/src/pool/PoolLendingV2.sol*		
Locations	*PoolLending._deposit L: 293 - 298* *PoolLendingV2._deposit L: 274 - 279*		

Detailed Issue

In the *PoolLending* and *PoolLendingV2* contracts, when lenders deposit, they receive *itpTokens* proportional to their deposit.

The contract calculates the itpPrice, which determines the proportion of the depositing token to the *itpTokenTotalSupply*, as shown in code snippet 17.1. This *itpPrice* is then used to calculate the itpTokens to be minted to lenders, as detailed in code snippet 17.2 L296 and 17.3 L277.

However, we've found a vulnerability where attackers can inflate the *itpPrice* to a level that prevents other lenders from receiving any *itpTokens*:

For instance, consider the scenario:

1. There are no deposits in the USDT pool initially.
2. **The attacker deposits 100,000e6 USDT** to the pool.
3. **The attacker then borrows 99,999e6 USDT** from the pool.
4. **The attacker** waits for a few blocks **and repays to close the loan**.
5. The contract charges the minimum interest with the repayment. The repayment is sent to the interest vault.
6. **The attacker calls the *claimAllInterest*** function to claim all collected interest from the interest vault.
   This should result in 0 claimableTokenInterest left in the interest vault.
   However, there will be some left. This is the result of a **rounding down error** in *itpPrice* calculation which is used in calculating the claimableToken as shown in code snippet 17.4. Since:
   itpPrice = (pTokenTotalSupply + InterestVault.claimableTokenInterest()) / itpTokenTotalSupply
   So the **fractional value will be truncated from the itpPrice**.

As a result, **there are 40,878 tokens left in the claimableTokenInterest** even though the only lender has claimed all his interest

7. The attacker can use this left claimableTokenInterest for his benefit.
   To use this as leverage to inflate itpPrice, **the attacker withdraws almost all of his USDT, leaving 1 USDT deposited.**
   Now the pTokenTotalSupply = 1
   itpTokenTotalSupply = 2
   claimableTokenInterest = 40878
   This results in itpPrice = 20439500000
8. Now the itpPrice is high enough to be easily manipulated, **the attacker can repeatedly deposit the maximum amount of USDT that will not mint any itpTokens.**
   For example, since itpAmount to be minted is calculated from:
   (depositAmount * PRECISION_UNIT) / itpPrice
   since, itpPrice = 20439500000
   to find the maximum amount of deposit that will not mint any itpToken, the formula is:
   (itpPrice / PRECISION_UNIT) - 1
   (20439500000 / 1e6) - 1 = 20,438
   so if the attacker deposits 20,438 tokens:
   the **minting itpAmount** = 20,438 * 1e6 / 20439500000 = 0.9 which is rounded down to **0**
9. This will cause the pTokenTotalSupply to increase without increasing itpTokenTotalSupply. Furthermore, each attack will reduce the required funds for the next one.
   For example, after depositing 20,438 tokens the result will be:
   pTokenTotalSupply = 20,439
   itpTokenTotalSupply = 2
   itpPrice = 30658500000
   claimableTokenInterest = 40,878
   Now the next maximum amount of deposit that will not mint any itpToken is:
   (itpPrice / PRECISION_UNIT) - 1
   = (30658500000 / 1e6) - 1
   = 30,657 tokens
10. **Since each attack will require fewer funds.** This will require less gas fee to perform the attack on a large scale**. If the attacker performs the 40th deposit,** the result will be:
    pTokenTotalSupply = 677973891734
    itpTokenTotalSupply = 2
    itpPrice = 338986966306000000
    claimableTokenInterest = 40,878

Now, the *itpPrice* is significantly high enough to **cause several unexpected behaviors** including

1. **Other lenders will not get any itpToken minted with their regular deposit amount.** For example, if lender A deposits 1,000e6 USDT to the pool, he will get 0 *itpToken* because (1,000e6 * 1e6) / 338986966306000000 = 0.
2. **The attacker can take interest from other lenders.** Since he controls all *itpToken* total supply, other lenders will not get any *itpToken* without depositing an overwhelming amount.

PoolBaseFunc.sol

```
function _getInterestTokenPrice() internal view returns (uint256) {
    if (itpTokenTotalSupply == 0) {
        return initialItpPrice;
    } else {
        return
            ((pTokenTotalSupply +
                IInterestVault(interestVaultAddress).claimableTokenInterest()) *
                PRECISION_UNIT) / itpTokenTotalSupply;
    }
}
```

Listing 17.1 The *_getInterestTokenPrice* function of the *PoolBaseFunc* contract

PoolLending.sol

```
function _deposit(
    address receiver,
    uint256 nftId,
    uint256 depositAmount
)
    internal
    returns (
        uint256 pMintAmount,
        uint256 atpMintAmount,
        uint256 itpMintAmount,
        uint256 ifpMintAmount
    )
{
    require(depositAmount > 0, "PoolLending/deposit-amount-is-zero");

    uint256 atpPrice = _getActualTokenPrice();
    uint256 itpPrice = _getInterestTokenPrice();
    uint256 ifpPrice = _getInterestForwPrice();

    //mint ip, atp, itp, ifp
    pMintAmount = _mintPToken(receiver, nftId, depositAmount);

    atpMintAmount = _mintAtpToken(
        receiver,
```

```
            nftId,
            ((depositAmount * PRECISION_UNIT) / atpPrice),
            atpPrice
        );

        itpMintAmount = _mintItpToken(
            receiver,
            nftId,
            ((depositAmount * PRECISION_UNIT) / itpPrice),
            itpPrice
        );
```

Listing 17.2 The *_deposit* function of the *PoolLending* contract

PoolLendingV2.sol

```
function _deposit(
    address receiver,
    uint256 nftId,
    uint256 depositAmount
)
    internal
    returns (
        uint256 pMintAmount,
        uint256 atpMintAmount,
        uint256 itpMintAmount,
        uint256 ifpMintAmount
    )
{
    require(depositAmount > 0, "PoolLending/deposit-amount-is-zero");

    uint256 atpPrice = _getActualTokenPrice();
    uint256 itpPrice = _getInterestTokenPrice();
    uint256 ifpPrice = _getInterestForwPrice();

    //mint ip, atp, itp, ifp
    pMintAmount = _mintPToken(receiver, nftId, depositAmount);

    atpMintAmount = _mintAtpToken(
        receiver,
        nftId,
        ((depositAmount * PRECISION_UNIT) / atpPrice),
        atpPrice
    );

    itpMintAmount = _mintItpToken(
        receiver,
        nftId,
        ((depositAmount * PRECISION_UNIT) / itpPrice),
```

```
        itpPrice
    );
```

Listing 17.3 The *_deposit* function of the *PoolLendingV2* contract

PoolLending.sol

```
function _claimTokenInterest(
    address receiver,
    uint256 nftId,
    uint256 claimAmount
) internal returns (WithdrawResult memory result) {
    uint256 itpPrice = _getInterestTokenPrice();
    PoolTokens storage tokenHolder = tokenHolders[nftId];

    uint256 claimableAmount;
    if (((tokenHolder.itpToken * itpPrice) / PRECISION_UNIT) >
tokenHolder.pToken) {
        claimableAmount =
            ((tokenHolder.itpToken * itpPrice) / PRECISION_UNIT) -
            tokenHolder.pToken;
    }
```

Listing 17.4 The *_claimTokenInterest* function of the *PoolLending* contract

Recommendations

We recommend the *FWX* team prevent this attack by keeping some tokens in the pool.

This can be done by either

1. **The FWX team must be the first depositor and deposit a fixed amount of tokens in the pool. Note that, this deposited token always needs to be in the pool** to significantly increase the tokens the attacker needs to inflate the *itpPrice*.
2. The protocol could **split a fixed amount of the first depositor's tokens into the deposit amount of address zero (similar to Uniswap's approach). This will ensure that a fixed amount of tokens is always in the pool**. Therefore, it will significantly increase the tokens the attacker needs to inflate the *itpPrice*.

For instance, consider a slightly different scenario where there are some initial tokens in the pool:

1. **There is an initial deposit of 100e6USDT to the pool.**
2. The attacker deposits 100,000e6 USDT to the pool.
3. The attacker then borrows 99,999e6 USDT from the pool.
4. The attacker waits for a few blocks and repays to close the loan.

5. The contract charges the minimum interest with the repayment. The repayment is sent to the interest vault.
6. The attacker calls claimAllInterest() to claim all collected interest from the interest vault.
7. The attacker withdraws almost all of his USDT, leaving 1 USDT deposited.
   Now the pTokenTotalSupply = 100000001
   itpTokenTotalSupply = 100000002

   claimableTokenInterest = 321250
   This results in itpPrice = 1003212

With 100e6 USDT in the pool prior to the attack, **the attacker can no longer inflate *itpPrice* with the same amount of funds (100,000e6 USDT)**.

Reassessment

The *FWX* team has acknowledged this issue. As a solution, *FWX* will initially provide liquidity with a significant amount locked in the pool permanently; this measure is designed to prevent potential inflation attacks.

No. 18	The itpBurnAmount Does Not Align With WithdrawAmount		
Risk	Medium	Likelihood	Medium
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/pool/PoolLending.sol* *contracts/src/pool/PoolLendingV2.sol*		
Locations	*PoolLending._withdraw L: 347* *PoolLendingV2._withdraw L: 328*		

Detailed Issue

In the *PoolLending* and *PoolLendingV2* contracts, when lenders withdraw, their *itpToken* is burnt according to the proportion of their withdrawal. This mechanism is designed to account for the interest that lenders should receive.

However, we found an issue caused by a rounding error as shown in code snippet 18.1 and 18.2, **this rounding down will cause lenders to have less *itpToken* burnt. As a result, lenders will be allowed to receive more *itpTokens*.**

For instance, consider a scenario where:

1. **Lender A deposits 100e6 tokens**, he receives 100e6 pTokens and itpTokens.
2. **Attacker deposits 100e6 tokens**, he receives 100e6 pTokens and itpTokens.
3. **A borrower borrows and repays**, this results in interest sent to the pool.
4. **InterestVault.claimableTokenInterest() = 100e6**.
5. itpPrice = (pTokenTotalSupply + InterestVault.claimableTokenInterest()) * PRECISION_UNIT / itpTokenTotalSupply.
   itpPrice = (200e6 + 100e6) * 1e6 / 200e6
   itpPrice = 1.5e6.
6. **Now if both attacker and lender A normally withdraws all tokens**, they should receive interest in a 50:50 manner. **Each lender should receive 100e6 / 2 = 50e6 tokens**.
7. **Attacker withdraws 1 wei USDT**.
8. Since, burnItp = (withdrawAmount * PRECISION_UNIT) / itpPrice
9. burnItp = (1 * 1e6) / 1.5e6
   = 0.xx which is rounded down to 0.
   burntItp = 0.
10. **The attacker repeats withdrawing until he has 1 wei USDT left in the pool**.

11. Now the attacker still has 100e6 *itpTokens*. When he **calls the claimAllInterest** function, the claimableAmount will be calculated as:
    As shown in code snippet 20.3 and 20.4:
    claimableAmount = ((tokenHolder.itpToken * itpPrice) / PRECISION_UNIT) - tokenHolder.pToken;
    claimableAmount = ((100e6 * 1.5e6) / 1e6) - 1
    claimableAmount = 1.49999999e8
12. itpBurnt = (claimAmount * PRECISION_UNIT) / itpPrice
    itpBurnt = 99.999999e6
13. **The attacker now receives almost all itpTokens from the pool**, this exceeds the expected amount he should receive which is 50e6.

There are practical considerations that make this attack challenging to execute on the live network, since **repeating step 10 will cost a huge amount of gas**. This makes the attack unprofitable if the attacker pays a normal gas price.

However, if the attacker is a miner then their gas price is zero, so this attack costs them nothing in gas.

PoolLending.sol

```
function _withdraw(
    address receiver,
    uint256 nftId,
    uint256 withdrawAmount
) internal returns (WithdrawResult memory) {
    // (...SNIPPET...)
    uint256 itpBurnAmount = _burnItpToken(
        receiver,
        nftId,
        (withdrawAmount * PRECISION_UNIT) / itpPrice,
        itpPrice
    );
```

Listing 18.1 The *_withdraw* function of the *PoolLending* contract

PoolLendingV2.sol

```
function _withdraw(
    address receiver,
    uint256 nftId,
    uint256 withdrawAmount
) internal returns (WithdrawResult memory) {
    // (...SNIPPET...)
    uint256 itpBurnAmount = _burnItpToken(
        receiver,
        nftId,
```

```
        (withdrawAmount * PRECISION_UNIT) / itpPrice,
        itpPrice
    );
```

Listing 18.2 The *_withdraw* function of the *PoolLendingV2* contract

PoolLending.sol

```
function _claimTokenInterest(
    address receiver,
    uint256 nftId,
    uint256 claimAmount
) internal returns (WithdrawResult memory result) {
    uint256 itpPrice = _getInterestTokenPrice();
    PoolTokens storage tokenHolder = tokenHolders[nftId];

    uint256 claimableAmount;
    if (((tokenHolder.itpToken * itpPrice) / PRECISION_UNIT) > tokenHolder.pToken) {
        claimableAmount =
            ((tokenHolder.itpToken * itpPrice) / PRECISION_UNIT) -
            tokenHolder.pToken;
    }

    claimAmount = MathUpgradeable.min(claimAmount, claimableAmount);
```

Listing 18.3 The *_claimTokenInterest* function of the *PoolLending* contract

PoolLendingV2.sol

```
function _claimTokenInterest(
    address receiver,
    uint256 nftId,
    uint256 claimAmount
) internal returns (WithdrawResult memory result) {
    uint256 itpPrice = _getInterestTokenPrice();
    PoolTokens storage tokenHolder = tokenHolders[nftId];

    uint256 claimableAmount;
    if (((tokenHolder.itpToken * itpPrice) / PRECISION_UNIT) > tokenHolder.pToken) {
        claimableAmount =
            ((tokenHolder.itpToken * itpPrice) / PRECISION_UNIT) -
            tokenHolder.pToken;
    }

    claimAmount = MathUpgradeable.min(claimAmount, claimableAmount);
```

Listing 18.4 The *_claimTokenInterest* function of the *PoolLendingV2* contract

Recommendations

The calculation of the *itpBurnAmount* involves an inevitable rounding error. The decision on who benefits from this rounding belongs to the *FWX* team.

Since the fractional value of *itpBurnAmount* is truncated rather than being rounded up**. We recommend that rounding *itpBurnAmount* should be prioritized to the other lenders over the attacker.**

However, the *FWX* team has the flexibility to adopt and adapt these recommendations based on their specific business requirements and priorities.

Reassessment

The *FWX* team has addressed this issue by improving the *_withdraw* function in the *PoolLending* and *PoolLendingV2* contracts. The team added a crucial requirement that ensures that every withdrawal results in a non-zero amount of itp tokens being burned.

PoolLending.sol

```
function _withdraw(
        address receiver,
        uint256 nftId,
        uint256 withdrawAmount
    ) internal returns (WithdrawResult memory) {

// (...SNIPPED...)

uint256 itpBurnAmount = _burnItpToken(
            receiver,
            nftId,
            (withdrawAmount * PRECISION_UNIT) / itpPrice,
            itpPrice
        );
        require((itpBurnAmount * itpPrice) / PRECISION_UNIT > 0,
"withdraw-amount-is-zero");

// (...SNIPPED...)
}
```

Listing 18.5 The improved *_withdraw* function of the *PoolLending* and *PoolLendingV2* contract

No. 19	Potential Over-Distribution Of Lending Bonuses		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Acknowledged
Associated Files	*contracts/src/pool/PoolLendingV2.sol* *contracts/src/pool/InterestVaultV2.sol*		
Locations	*PoolLendingV2._claimTokenInterest* *InterestVaultV2._withdrawTokenInterest*		

Detailed Issue

We found that an over-distributed *bonusAmount* value is used in the claim interest process through the *claimTokenInterest* function of the *PoolLendingV2* contract. Consider the following scenario:

Assumptions:

- The **APH Pool has 2 lenders with equal principal value before interest is accrued**, giving them **equal power to claim interest**.
- Initial state for demonstration:
  - Total Interest accrued = **300 * 1e18**
  - *heldTokenInterest* = (300 * 1e18) * 10% = **30 * 1e18**
  - *claimableInterest* = 300 * 1e18 - *heldTokenInterest* = **270 * 1e18**
  - *interestBonusLending* = **11.12%**
  - Each lender's claimable interest = (300 * 1e18)/2 = **135 * 1e18**

Steps:

1. **First lender claims all their interest:**
   - *claimableAmount* = **135 * 1e18**
   - *bonusAmount* = (135 * 1e18) * *interestBonusLending%* = **15.012 * 1e18**
   - *profitAmount* = (135 * 1e18) * 10 / (100 - 10) = **15 * 1e18**
   - The *bonusAmount* (15.012 * 1e18) is greater than the *profitAmount* (15 * 1e18)
   - Actual *profitAmount* = 15 * 1e18 - **min(**15.012 * 1e18, **15 * 1e18)** = **0**

2. **Value passed to withdrawTokenInterest function:**
    - *IInterestVaultV2(interestVaultAddress).withdrawTokenInterest(claimable: claimableAmount, bonus:* ***15.012*** *** ***1e18****, profit: 0);*
    - The *bonusAmount* (15.012 * 1e18) is greater than the *profitAmount* (0), and as it is less than the *heldTokenInterest* (30 * 1e18), the full *bonusAmount* is claimed and transferred back to the claimer.

3. **Remaining *heldTokenInterest*:**
    - Remaining ***heldTokenInterest*** = (30 - 15.012) * 1e18 = **14.988 * 1e18**

In this scenario, when the second lender claims their interest, they will only receive 14.988 * 1e18 tokens as a bonus/profit, despite having the same claim power.

This results in an unfair distribution where the first lender receives more than their fair share of the bonus, and the second lender receives less than their rightful profit.

PoolLendingV2.sol

```
function claimTokenInterest(
    uint256 nftId,
    uint256 claimAmount
)
    external
    nonReentrant
    whenFuncNotPaused(msg.sig)
    settleForwInterest
    returns (WithdrawResult memory result)
{
    nftId = _getUsableToken(msg.sender, nftId);
    result = _claimTokenInterest(msg.sender, nftId, claimAmount);
    _transferFromOut(
        interestVaultAddress,
        msg.sender,
        tokenAddress,
        result.tokenInterest + result.tokenInterestBonus
    );
    return result;
}
```

Listing 19.1 The *claimTokenInterest* function of the *PoolLendingV2* contract

PoolLendingV2.sol

```
function _claimTokenInterest(
    address receiver,
    uint256 nftId,
    uint256 claimAmount
) internal returns (WithdrawResult memory result) {
    uint256 itpPrice = _getInterestTokenPrice();
    PoolTokens storage tokenHolder = tokenHolders[nftId];

    uint256 claimableAmount;
    if (((tokenHolder.itpToken * itpPrice) / PRECISION_UNIT) > tokenHolder.pToken) {
        claimableAmount =
            ((tokenHolder.itpToken * itpPrice) / PRECISION_UNIT) -
            tokenHolder.pToken;
    }

    claimAmount = MathUpgradeable.min(claimAmount, claimableAmount);

    uint256 burnAmount = _burnItpToken(
        receiver,
        nftId,
        (claimAmount * PRECISION_UNIT) / itpPrice,
        itpPrice
    );
    uint256 bonusAmount = (claimAmount * _getPoolRankInfo(nftId).interestBonusLending) /
            WEI_PERCENT_UNIT;

    uint256 feeSpread = IAPHCore(coreAddress).feeSpread();
    uint256 profitAmount = ((claimAmount * feeSpread) / (WEI_PERCENT_UNIT - feeSpread));
    profitAmount -= MathUpgradeable.min(bonusAmount, profitAmount);

    (claimAmount, bonusAmount, profitAmount) = IInterestVaultV2(interestVaultAddress)
            .withdrawTokenInterest(claimAmount, bonusAmount, profitAmount);

    emit ClaimTokenInterest(receiver, nftId, claimAmount, bonusAmount, burnAmount);

    result.tokenInterest = claimAmount;
    result.itpTokenBurn = burnAmount;
    result.tokenInterestBonus = bonusAmount;
}
```

Listing 19.2 The *_claimTokenInterest* function of the *PoolLendingV2* contract

InterestVaultV2.sol

```
function _withdrawTokenInterest(
    uint256 claimable,
    uint256 bonus,
    uint256 profit
) internal returns (uint256 claimedInterest, uint256 claimedBonus, uint256
claimedProfit) {
    claimedInterest = Math.min(claimable, claimableTokenInterest);
    if (bonus > heldTokenInterest) {
        claimedBonus = heldTokenInterest;
        claimedProfit = 0;
    } else if (bonus + profit > heldTokenInterest) {
        claimedBonus = bonus;
        claimedProfit = heldTokenInterest - bonus;
    } else {
        claimedBonus = bonus;
        claimedProfit = profit;
    }

    claimableTokenInterest -= claimedInterest;
    heldTokenInterest -= claimedBonus + claimedProfit;
    actualTokenInterestProfit += profit;
    cumulativeTokenInterestProfit += profit;

    emit WithdrawTokenInterest(msg.sender, claimedInterest, claimedBonus,
claimedProfit);
}
```

Listing 19.3 The *_withdrawTokenInterest* function of the *InterestVaultV2* contract

Recommendations

We recommend implementing logic to ensure that the *bonusAmount* is appropriately bounded by the *profitAmount* to avoid situations where the bonus exceeds the profit.

Reassessment

The *FWX* statement has acknowledged with the statement:

"The FWX team has verified that the interestBonusLending will be below 11.11%. We recognize that certain lenders may not claim their entire interest bonus, leaving behind minimal amounts typically considered negligible."

No. 20	Bypassing The checkStakingAmountSufficient When The PriceFeed Is Paused		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Acknowledged
Associated Files	*contracts/src/pool/PoolLending.sol* *contracts/src/pool/PoolLendingV2.sol* *contracts/src/core/APHCore.sol*		
Locations	*PoolLending.deposit L: 83* *PoolLendingV2.deposit L: 71* *APHCore.checkStakingAmountSufficient L: 167 - 202*		

Detailed Issue

The price feed of the protocol could pause from the global config represented in line 162 in code snippet 20.1. **It always returned 0 for rate and precision when paused**.

We noticed that the *deposit* function (code snippet 20.2) of the *PoolLending* and *PoolLendingV2* contracts will validate the deposit amount by invoking the *checkStakingAmountSufficient* function of the *APHCore* contract (code snippet 20.2) to ensure that the staking amount of the NFT ID had sufficient for a given deposit amount.

Within the *checkStakingAmountSufficient* function, it retrieves the rate from the price feed to calculate the *requireBalance* for validating at line 197 in the code snippet 20.3

The issue occurs when the price feed has been paused, while the rate returned from the *queryRateUSD* function always returns 0, which allows validation of the *checkStakingAmountSufficient* function always passes. **Consequently, users can bypass the *checkStakingAmountSufficient* function when depositing**.

PriceFeeds.sol

```
161 function _queryRateUSD(address token) internal view returns (uint256 rate,
    uint256 precision) {
162     if (pricesFeeds[token] == address(0) || globalPricingPaused) return (0, 0);
163     AggregatorV2V3Interface _Feed = AggregatorV2V3Interface(pricesFeeds[token]);
164     (, int256 answer, , uint256 updatedAt, ) = _Feed.latestRoundData();
165     require(answer > 0, "PriceFeed/price-must-be-greater-than-zero");
166     rate = uint256(answer);
167     uint256 decimal = _Feed.decimals();
168
```

```
    rate = (rate * WEI_PRECISION) / (10 ** decimal);
    precision = WEI_PRECISION;

    require(block.timestamp - updatedAt < stalePeriod[token],
"PriceFeed/price-is-stale");
}
```

Listing 20.1 The *_queryRateUSD* function of the *PriceFeeds* contract

PoolLendingV2.sol

```
function deposit(
    uint256 nftId,
    uint256 depositAmount
)
    external
    payable
    nonReentrant
    whenFuncNotPaused(msg.sig)
    settleForwInterest
    returns (uint256 mintedP, uint256 mintedAtp, uint256 mintedItp, uint256
mintedIfp)
{
    require(
        tokenAddress == wethAddress || msg.value == 0,
        "PoolLending/no-support-transfering-ether-in"
    );
    nftId = _getUsableToken(msg.sender, nftId);

    if (tokenHolders[nftId].pToken != 0) {
        require(lenders[nftId].rank == _getNFTRank(nftId),
"PoolLending/nft-rank-not-match");
    } else {
        lenders[nftId].rank = _getNFTRank(nftId);
        lenders[nftId].updatedTimestamp = uint64(block.timestamp);
    }
    IAPHCore core = IAPHCore(coreAddress);
    core.checkStakingAmountSufficient(nftId, depositAmount, tokenAddress);

    _transferFromIn(msg.sender, address(this), tokenAddress, depositAmount);
    (mintedP, mintedAtp, mintedItp, mintedIfp) = _deposit(msg.sender, nftId,
depositAmount);
}
```

Listing 20.2 The example *deposit* function of the *PoolLendingV2* contract

APHCore.sol

```
function checkStakingAmountSufficient(
    uint256 nftId,
    uint256 newAmount,
    address tokenAddress
) external view returns (uint256) {
    IPriceFeed priceFeed = IPriceFeed(priceFeedAddress);
    {
        (uint256 rate, ) = priceFeed.queryRateUSD(tokenAddress);
        newAmount = (newAmount * rate) / tokenPrecisionUnit[tokenAddress]; // 1e18
    }

    uint256 totalUSDOfUserLent; // 1e18
    for (uint i = 0; i < poolList.length; i++) {
        IAPHPool pool = IAPHPool(poolList[i]);
        uint256 pToken = pool.balancePTokenOf(nftId);
        (uint256 rate, ) = priceFeed.queryRateUSD(pool.tokenAddress());
        totalUSDOfUserLent =
            totalUSDOfUserLent +
            ((pToken * rate) / tokenPrecisionUnit[pool.tokenAddress()]);
    }

    totalUSDOfUserLent += newAmount;
    StakePoolBase.StakeInfo memory stakeInfo = IStakePool(
        IMembership(membershipAddress).currentPool()
    ).getStakeInfo(nftId);

    uint256 currentStakedBalance = stakeInfo.stakeBalance;
    uint256 requireBalance = ((totalUSDOfUserLent * forwStakingMultiplier) / WEI_UNIT);

    require(
        currentStakedBalance >= requireBalance,
        "APHCore/stake-not-matched-minimum-requirement"
    );

    return requireBalance;
}
```

Listing 20.3 The *checkStakingAmountSufficient* function of the *APHCore* contract

Recommendations

We recommend validating the returned *rate* from the price feed to ensure the price feed is not paused before use as shown in the code snippet below.

APHCore.sol

```
function checkStakingAmountSufficient(
    uint256 nftId,
    uint256 newAmount,
    address tokenAddress
) external view returns (uint256) {
    IPriceFeed priceFeed = IPriceFeed(priceFeedAddress);
    {
        (uint256 rate, ) = priceFeed.queryRateUSD(tokenAddress);
        require(rate != 0, "APHCore/price-feed-paused");
        newAmount = (newAmount * rate) / tokenPrecisionUnit[tokenAddress]; // 1e18
    }

    uint256 totalUSDOfUserLent; // 1e18
    for (uint i = 0; i < poolList.length; i++) {
        IAPHPool pool = IAPHPool(poolList[i]);
        uint256 pToken = pool.balancePTokenOf(nftId);
        (uint256 rate, ) = priceFeed.queryRateUSD(pool.tokenAddress());
        totalUSDOfUserLent =
            totalUSDOfUserLent +
            ((pToken * rate) / tokenPrecisionUnit[pool.tokenAddress()]);
    }

    totalUSDOfUserLent += newAmount;
    StakePoolBase.StakeInfo memory stakeInfo = IStakePool(
        IMembership(membershipAddress).currentPool()
    ).getStakeInfo(nftId);

    uint256 currentStakedBalance = stakeInfo.stakeBalance;
    uint256 requireBalance = ((totalUSDOfUserLent * forwStakingMultiplier) / WEI_UNIT);

    require(
        currentStakedBalance >= requireBalance,
        "APHCore/stake-not-matched-minimum-requirement"
    );

    return requireBalance;
}
```

Listing 20.4 The improved *checkStakingAmountSufficient* function of the *APHCore* contract

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *FWX* team has acknowledged this issue as it is an intentional design.

No. 21	Division By Zero When The Global PriceFeed Is Paused		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/utils/PriceFeed.sol*		
Locations	*PriceFeed._queryRate L: 152*		

Detailed Issue

The *_queryRate* function retrieves the source and destination token rates from the *_queryRateUSD* function before calculating the rate between the source and destination tokens. **We identified an issue where, if global pricing is paused by the manager, the rate returned by the *_queryRateUSD* function is zero** (L162 in code snippet below). **This results in a division by zero when the *_queryRate* function attempts to calculate the rate on line 152, causing the operation to revert.**

PriceFeed.sol

```
136 function _queryRate(
137      address sourceToken,
138      address destToken
139 ) internal view returns (uint256 rate, uint256 precision) {
140      uint256 sourceRate;
141      uint256 destRate;
142      if (sourceToken != destToken) {
143          (sourceRate, ) = _queryRateUSD(sourceToken);
144          (destRate, ) = _queryRateUSD(destToken);
145
146          if (sourceRate == 0 || destRate == 0) {
147              rate = 0;
148          } else {
149              rate = (sourceRate * WEI_PRECISION) / destRate;
150          }
151
152          rate = (sourceRate * WEI_PRECISION) / destRate;
153
154          precision = _getDecimalPrecision(sourceToken, destToken);
155      } else {
156          rate = WEI_PRECISION;
157          precision = WEI_PRECISION;
158      }
```

```
159 }
160 
161 function _queryRateUSD(address token) internal view returns (uint256 rate, uint256 precision) {
162     if (pricesFeeds[token] == address(0) || globalPricingPaused) return (0, 0);
163         AggregatorV2V3Interface _Feed = AggregatorV2V3Interface(pricesFeeds[token]);
164         (, int256 answer, , uint256 updatedAt, ) = _Feed.latestRoundData();
165         require(answer > 0, "PriceFeed/price-must-be-greater-than-zero");
166         rate = uint256(answer);
167         uint256 decimal = _Feed.decimals();
168 
169         rate = (rate * WEI_PRECISION) / (10 ** decimal);
170         precision = WEI_PRECISION;
171 
172         require(block.timestamp - updatedAt < stalePeriod[token], "PriceFeed/price-is-stale");
173 }
```

Listing 21.1 The *_queryRate and _queryRateUSD* functions of the *PriceFeed* contract

Recommendations

We recommend **removing the redundant line *rate* = *(sourceRate* * *WEI_PRECISION)* / *destRate;*** from line 152, as the rate calculation is already handled within the *_queryRate* function (L146-150 in code snippet 21.1).

Reassessment

The *FWX* team adopted our recommendation and fixed this issue.

No. 22	Overpayment For requiredAmount Not Be Refunded		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/core/CoreLiquidateWithoutSwap.sol*		
Locations	*CoreLiquidateWithoutSwap._liquidateLoanWithoutSwap L: 249 - 256* *CoreLiquidateWithoutSwap._setPositionAndCheckIncreaseBalance L: 598*		

Detailed Issue

Both functions *_liquidateLoanWithoutSwap* and *_setPositionAndCheckIncreaseBalance* calls *APHCoreLiquidateCallee.liquidatePositionWithoutSwapCall* to get the required funds to perform *_liquidateLoanWithoutSwap, _liquidateLongWithoutSwap* and *_liquidateShortWithoutSwap.*

However, we found that if *APHCoreLiquidateCallee* sends more funds than requested, The extra amount of funds will make *repayInterest* in *_liquidateLoanWithoutSwap* higher than expected as shown in code snippet 22.1.

On the other hand, the extra funds in *_setPositionAndCheckIncreaseBalance* function will be stuck in the *APHCore* contract, since the contract allows the sent funds to be higher than expected as shown in code snippet 22.2.

CoreLiquidateWithoutSwap.sol

```
147 function _liquidateLoanWithoutSwap(
148     uint256 nftId,
149     uint256 loanId,
150     address to,
151     bytes calldata data
152 )
153     internal
154     returns (
155         uint256 repayBorrow,
156         uint256 repayInterest,
157         uint256 collateralToLiquidator,
158         uint256 leftOverCollateral
159     )

241         {
```

```
        uint256 previousBalance =
IERC20(loan.borrowTokenAddress).balanceOf(address(this));
        IAPHCoreLiquidateCallee(to).liquidateLoanWithoutSwapCall(
            msg.sender,
            (repayInterest + repayBorrow),
            collateralToLiquidator,
            data
        );

        uint256 increaseBalance =
IERC20(loan.borrowTokenAddress).balanceOf(address(this)) -
            previousBalance;

            require(
                increaseBalance >= (repayInterest + repayBorrow),
                "CoreBorrowing/insufficient-require-amount"
            );
            repayInterest = increaseBalance - repayBorrow;
        }
```

Listing 22.1 The *_liquidateLoanWithoutSwap* function of the *CoreLiquidateWithoutSwap* contract

CoreLiquidateWithoutSwap.sol

```
function _setPositionAndCheckIncreaseBalance(
    PositionState storage posState,
    Position memory pos,
    APHLibrary.LiquidatePositionWithoutSwapParams memory params,
    uint256 requiredAmount,
    uint256 collaToLiquidator
) internal returns (uint256 increaseBalance) {
    posState.active = false;
    {
        uint256 previousBalance =
IERC20(pos.borrowTokenAddress).balanceOf(address(this));
        IAPHCoreLiquidateCallee(params.to).liquidatePositionWithoutSwapCall(
            msg.sender,
            requiredAmount,
            posState.isLong ? pos.contractSize : 0,
            collaToLiquidator,
            params.data
        );
        increaseBalance =
            IERC20(pos.borrowTokenAddress).balanceOf(address(this)) -
            previousBalance;
    }
    require(increaseBalance >= requiredAmount,
"CoreBorrowing/insufficient-require-amount");
```

```
    return increaseBalance;
}
```

Listing 22.2 The *_setPositionAndCheckIncreaseBalance* function of the *CoreLiquidateWithoutSwap* contract

Recommendations

We recommend checking the sent fund to be equal to the requested funds. This recommendation is illustrated in code snippet 22.3 L253 and 22.4 L598.

CoreLiquidateWithoutSwap.sol

```
147 function _liquidateLoanWithoutSwap(
148     uint256 nftId,
149     uint256 loanId,
150     address to,
151     bytes calldata data
152 )
153     internal
154     returns (
155         uint256 repayBorrow,
156         uint256 repayInterest,
157         uint256 collateralToLiquidator,
158         uint256 leftOverCollateral
159     )

241         {
242         uint256 previousBalance =
243 IERC20(loan.borrowTokenAddress).balanceOf(address(this));
244         IAPHCoreLiquidateCallee(to).liquidateLoanWithoutSwapCall(
245             msg.sender,
246             (repayInterest + repayBorrow),
247             collateralToLiquidator,
248             data
249         );
250
251         uint256 increaseBalance =
    IERC20(loan.borrowTokenAddress).balanceOf(address(this)) -
            previousBalance;
252
253             require(
                increaseBalance == (repayInterest + repayBorrow),
                "CoreBorrowing/insufficient-require-amount"
254             );
255             repayInterest = increaseBalance - repayBorrow;
256         }
```

Listing 22.3 The improved *_liquidateLoanWithoutSwap* function in the *CoreLiquidateWithoutSwap* contract

CoreLiquidateWithoutSwap.sol

```
function _setPositionAndCheckIncreaseBalance(
    PositionState storage posState,
    Position memory pos,
    APHLibrary.LiquidatePositionWithoutSwapParams memory params,
    uint256 requiredAmount,
    uint256 collaToLiquidator
) internal returns (uint256 increaseBalance) {
    posState.active = false;
    {
        uint256 previousBalance =
IERC20(pos.borrowTokenAddress).balanceOf(address(this));
        IAPHCoreLiquidateCallee(params.to).liquidatePositionWithoutSwapCall(
            msg.sender,
            requiredAmount,
            posState.isLong ? pos.contractSize : 0,
            collaToLiquidator,
            params.data
        );
        increaseBalance =
            IERC20(pos.borrowTokenAddress).balanceOf(address(this)) -
            previousBalance;
    }
    require(increaseBalance == requiredAmount,
"CoreBorrowing/insufficient-require-amount");
    return increaseBalance;
}
```

Listing 22.4 The improved *_setPositionAndCheckIncreaseBalance* function in the *CoreLiquidateWithoutSwap* contract

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *FWX* team adopted our recommendation and fixed this issue.

No. 23	Potential Denial Of Liquidate On Blacklisted Loaner		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Acknowledged
Associated Files	*contracts/src/core/CoreBorrowing.sol* *contracts/src/core/CoreLiquidateWithoutSwap.sol*		
Locations	*CoreBorrowing.liquidate L: 243* *CoreLiquidateWithoutSwap.liquidateLoanWithoutSwap L: 62*		

Detailed Issue

The protocol allows the liquidator to call the *liquidate* function of the *CoreBorrowing* or *liquidateLoanWithoutSwap* function of the *CoreLiquidateWithoutSwap* contract when the loan reaches the liquidation criteria, where remain collateral will be divided into the *bountyfee* for the liquidator, *bountyToProtocol* for the protocol, and the *leftOverCollateral* which is the leftover back to the loaner respectively (L234, 236, and 243 in the code snippet below).

However, we noticed the allowed collateral could be any *ERC20* by the admin listed. The point we are concerned about is that some of *ERC20* has a blacklist feature that prevents blacklisted addresses from sending out or receiving tokens or both e.g., *USDC*. (code snippet 23.3)

The issue occurs when the liquidator calls the liquidation function to liquidate the loan while the loaner is blacklisted from the collateral token. Consequently, the transaction always reverts and directly adversely affects the liquidator and protocol.

CoreBorrowing.sol

```
201 function liquidate(
202     uint256 loanId,
203     uint256 nftId
204 )
205     external
206     whenFuncNotPaused(msg.sig)
207     nonReentrant
208     returns (
209         uint256 repayBorrow,
210         uint256 repayInterest,
211         uint256 bountyReward,
212         uint256 bountyToProtocol,
213         uint256 leftOverCollateral
```

```
    )
{
    Loan storage loan = loans[nftId][loanId];
    (
        repayBorrow,
        repayInterest,
        bountyReward,
        bountyToProtocol,
        leftOverCollateral
    ) = _liquidate(loanId, nftId);

    IERC20Upgradeable(loan.borrowTokenAddress).safeTransfer(
        assetToPool[loan.borrowTokenAddress],
        repayBorrow
    );
    IERC20Upgradeable(loan.borrowTokenAddress).safeTransfer(
        _getInterestVault(assetToPool[loan.borrowTokenAddress]),
        repayInterest
    );

    _transferOut(msg.sender, loan.collateralTokenAddress, bountyReward);
    if (bountyToProtocol > 0) {
        _safeTransfer(loan.collateralTokenAddress, feeVaultAddress, bountyToProtocol);
        IFeeVault(feeVaultAddress).settleFeeProfitAndFeeAuction(
            loan.collateralTokenAddress,
            bountyToProtocol,
            0
        );
    }
    _transferOut(_getTokenOwnership(nftId), loan.collateralTokenAddress, leftOverCollateral);
}
```

Listing 23.1 The *liquidate* function of the *CoreBorrowing* contract

AssetHandlerUpgradeable.sol

```
function _transferOut(address to, address token, uint256 amount) internal {
    if (amount == 0) {
        return;
    }
    if (token == wethAddress) {
        IWethERC20Upgradeable(wethAddress).safeTransfer(wethHandler, amount);
        IWethHandler(payable(wethHandler)).withdrawETH(to, amount);
    } else {
        IERC20Upgradeable(token).safeTransfer(to, amount);
    }
}
```

Listing 23.2 The *_transferOut* function of the *AssetHandlerUpgradeable* contract

FiatTokenV1.sol

```
/**
 * @notice Transfers tokens from the caller.
 * @param to    Payee's address.
 * @param value Transfer amount.
 * @return True if the operation was successful.
 */
function transfer(address to, uint256 value)
    external
    override
    whenNotPaused
    notBlacklisted(msg.sender)
    notBlacklisted(to)
    returns (bool)
{
    _transfer(msg.sender, to, value);
    return true;
}
```

Listing 23.3 The *transfer* function in the *FiatTokenV1* contract (USDC) on the *Ethereum* chain
https://etherscan.io/address/0x43506849d7c04f9138d1a2050bbf3a0c054402dd#code

Recommendations

There is no recommendation code for this issue as it might break the contract functionality and require a decision from the *FWX* team in terms of business and protocol's core functionality.

However, **we recommend the *FWX* team apply a Pull Payment instead of the Push Payment to return leftover collateral to the loaner** or **allow only the *ERC20* collateral that does not block a transfer process to ensure the liquidation process will not be denied by the external factor**.

Reassessment

The *FWX* team has acknowledged this issue.

No. 24	Avoid Using block.number On Some L2 Chains		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Acknowledged
Associated Files	*contracts/src/core/CoreSetting.sol* *contracts/src/core/CoreBaseFunc.sol*		
Locations	*CoreSetting._setForwLendingDistributionPerBlock L: 235, 243* *CoreBaseFunc._settleForwInterest L: 37, 39, 45, 48, 58, 63*		

Detailed Issue

The *CoreSetting* contract and *CoreBaseFunc* contract uses *block.number* in various locations as shown in code snippet 24.1, 24.2 and 24.3.

block.number means different things on different L2s: **On Optimism, *block.number* is the L2 block number, but on Arbitrum, it's the L1 block number**.

Furthermore, L2 block numbers often occur much more frequently than L1 block numbers (any may even occur on a per-transaction basis), **so using block numbers for timing results in inconsistencies**.

CoreSetting.sol

```
250 function _setForwLendingDistributionPerBlock(
251     address _poolAddress,
252     uint256 _amount,
253     uint256 _targetBlock
254 ) internal {
255     require(poolToAsset[_poolAddress] != address(0),
"CoreSetting/pool-is-not-exist");
256
257     if (_targetBlock == 0) {
258         forwLendingDistributionPerBlock[_poolAddress] = _amount;
259
260         nextForwLendingDistributionPerBlock[_poolAddress].amount = 0;
261         nextForwLendingDistributionPerBlock[_poolAddress].targetBlock = 0;
262     } else {
263         require(_targetBlock >= block.number, "CoreSetting/error");
264
265         nextForwLendingDistributionPerBlock[_poolAddress].amount = _amount;
266         nextForwLendingDistributionPerBlock[_poolAddress].targetBlock =
_targetBlock;
```

```
        for (uint i = 0; i < poolList.length; i++) {
            lastSettleForw[poolList[i]] = block.number;
        }
    }

    emit SetForwLendingDistributionPerBlock(msg.sender, _amount, _targetBlock);
}
```

Listing 24.1 The *_setForwLendingDistributionPerBlock* function of the *CoreSetting* contract

CoreSetting.sol

```
function registerNewPool(
    address _poolAddress,
    uint256 _amount,
    uint256 _targetBlock
) external onlyConfigTimelockManager {
    require(poolToAsset[_poolAddress] == address(0),
"CoreSetting/pool-is-already-exist");

    address assetAddress = IAPHPool(_poolAddress).tokenAddress();

    for ((uint256 i, uint256 n) = (0, routers.length); i < n; i++) {
        if (routers[i] == address(0)) break;
        _approveForRouter(assetAddress, i, type(uint256).max);
    }

    poolToAsset[_poolAddress] = assetAddress;
    assetToPool[assetAddress] = _poolAddress;
    {
        uint256 precision = IERC20Metadata(assetAddress).decimals();
        tokenPrecisionUnit[assetAddress] = 10 ** precision;
    }
    swapableToken[assetAddress] = true;
    poolList.push(_poolAddress);

    lastSettleForw[_poolAddress] = block.number;

    _setForwLendingDistributionPerBlock(_poolAddress, _amount, _targetBlock);

    emit RegisterNewPool(msg.sender, _poolAddress);
}
```

Listing 24.2 The *registerNewPool* function of the *CoreSetting* contract

CoreBaseFunc.sol

```
function _settleForwInterest() internal returns (uint256 forwAmount) {
    if (lastSettleForw[msg.sender] != 0) {
        uint256 targetBlock =
nextForwLendingDistributionPerBlock[msg.sender].targetBlock;
        uint256 newForwLendingDistributionPerBlock =
nextForwLendingDistributionPerBlock[
            msg.sender
        ].amount;

        if (targetBlock != 0) {
            if (targetBlock >= block.number) {
                forwAmount =
                    (block.number - lastSettleForw[msg.sender]) *
                    forwLendingDistributionPerBlock[msg.sender];
            } else {
                forwAmount =
                    ((targetBlock - lastSettleForw[msg.sender]) *
                        forwLendingDistributionPerBlock[msg.sender]) +
                    ((block.number - targetBlock) *
newForwLendingDistributionPerBlock);
            }

            if (targetBlock <= block.number) {
                forwLendingDistributionPerBlock[
                    msg.sender
                ] = newForwLendingDistributionPerBlock;
                nextForwLendingDistributionPerBlock[
                    msg.sender
                ] = NextForwLendingDistributionPerBlock(0, 0);
            }
        } else {
            forwAmount =
                (block.number - lastSettleForw[msg.sender]) *
                forwLendingDistributionPerBlock[msg.sender];
        }
    }

    lastSettleForw[msg.sender] = block.number;
```

Listing 24.3 The *_settleForwInterest* function of the *CoreBaseFunc* contract

Recommendations

We recommend the *FWX* team to ensure that *block.number* on the deployed chain behaves as expected.
If it does not work as expected, we recommend the *FWX* to ensure that related values such as *forwLendingDistributionPerBlock* meet the business requirement.

Reassessment

The FWX team has acknowledged this issue with the statement:

"*We confirm that we have selected a chain where the block number aligns with our business flow.*"

No. 25	Price Diff Prevention Is Susceptible To Price Manipulation		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Acknowledged
Associated Files	*contracts/src/core/CoreFutureBaseFunc.sol* *contracts/src/utils/FwxAggregator.sol*		
Locations	*CoreFutureBaseFunc._getUnrealizedPNL L:163* *FwxAggregator._selectRouterWithReserves L: 75 - 96* *FwxAggregator._getAnswer L: 169 - 179*		

Detailed Issue

The hedging protocol coordinately uses the *Decentralized Exchange (DEX)* and the *Oracle Price Feed* to prevent the risk of the single source price impact as shown in the code snippet below. **To elaborate, the invoking of will check the price between *Oracle Price Feed* and *DEX* should not exceed the configured *maxOraclePriceDiffPercent*** first before allowing it to open, close, and liquidate position.

However, the *_queryRateUSD* function of the *PriceFeeds* contract (code snippet 25.2) is an internal call from the *queryRate* invoking (L157 in the code snippet 25.1) that allows utilizing whatever price feed contract that follows the *AggregatorV2V3Interface* interface (L163 in the code snippet 25.2).

We noticed that the *FwxAggregator* contract has derived the *AggregatorV2V3Interface* (code snippet 25.3) but the price is determined by the *Decentralized Exchange (DEX)* reserve instead of the *Oracle Price Feed*.

The root cause of this issue is that using the two *Decentralized Exchange (DEX)* instead of coordinately using *Decentralized Exchange (DEX)* and the *Oracle Price Feed* which may lead to susceptibility to price manipulation from the low liquidity *DEX*.

CoreFutureBaseFunc.sol

```
107 function _getUnrealizedPNL(
108     uint256 nftId,
109     bytes32 pairByte,
110     bool checkPriceDiff
111 ) internal returns (int256 pnl, uint256[] memory amounts, uint256 rate, uint256
    swapFee) {
112     address router;
113     Pair memory pair = pairs[pairByte];
114     Position memory pos = positions[nftId][pairByte];
```

```
    PositionState memory posState = positionStates[nftId][pos.id];

    {
        if (posState.isLong) {
            (amounts, swapFee, router) = _getAmountsWithRouterSelection(
                false,
                false,
                pairByte,
                pos.contractSize,
                pair.pair1,
                pair.pair0,
                0,
                0
            );
            rate = (amounts[1] * tokenPrecisionUnit[pair.pair1]) / amounts[0];
            pnl =
                ((int256(rate) - int256(pos.entryPrice)) * int256(pos.contractSize)) /
                int256(tokenPrecisionUnit[pair.pair1]);
        } else {
            uint256 swapAmount = ((pos.borrowAmount *
                (WEI_PERCENT_UNIT + _getNFTRankInfo(nftId).tradingFee)) / WEI_PERCENT_UNIT) +
                _calculateFutureInterest(nftId, pairByte);

            (amounts, swapFee, router) = _getAmountsWithRouterSelection(
                true,
                false,
                pairByte,
                swapAmount,
                pair.pair0,
                pair.pair1,
                0,
                0
            );
            rate = (amounts[0] * tokenPrecisionUnit[pair.pair1]) / amounts[1];
            pnl =
                ((int256(pos.entryPrice) - int256(rate)) * int256(pos.borrowAmount)) /
                int256(tokenPrecisionUnit[pair.pair1]);
        }

        SwapConfig memory swapConfig = swapConfigs[router][pairByte];
        if (checkPriceDiff) {
            uint256 UNDERLYING_PRECISION = tokenPrecisionUnit[pair.pair1];
            (uint256 oracleRate, uint256 precision) = IPriceFeed(priceFeedAddress).queryRate(
                pair.pair1,
                pair.pair0
            );
            oracleRate = (oracleRate * UNDERLYING_PRECISION) / precision;
```

```
            require(
                _checkPriceDiff(oracleRate, rate, swapConfig.maxOraclePriceDiffPercent),
                "CoreTrading/price-diff-too-high"
            );
        }
    }
}
```

Listing 25.1 The *_getUnrealizedPNL* function of the *CoreFutureBaseFunc* contract

PriceFeeds.sol

```
function _queryRateUSD(address token) internal view returns (uint256 rate, uint256 precision) {
    if (pricesFeeds[token] == address(0) || globalPricingPaused) return (0, 0);
    AggregatorV2V3Interface _Feed = AggregatorV2V3Interface(pricesFeeds[token]);
    (, int256 answer, , uint256 updatedAt, ) = _Feed.latestRoundData();
    require(answer > 0, "PriceFeed/price-must-be-greater-than-zero");
    rate = uint256(answer);
    uint256 decimal = _Feed.decimals();

    rate = (rate * WEI_PRECISION) / (10 ** decimal);
    precision = WEI_PRECISION;

    require(block.timestamp - updatedAt < stalePeriod[token], "PriceFeed/price-is-stale");
}
```

Listing 25.2 The *_queryRateUSD* function of the *PriceFeeds* contract

FwxAggregator.sol

```
contract FwxAggregator is AggregatorV2V3Interface {

    // (...SNIPPED...)

    function _selectRouterWithReserves()
        private
        view
        returns (address router, uint112 r0, uint112 r1)
    {
        for (uint256 i = 0; i < routers.length; i++) {
            IPancakePair pair = _getPair(routers[i], token0, token1);
            if (address(pair) == address(0)) continue;
            bool isSameToken0 = pair.token0() == token0;
            (uint112 _r0, uint112 _r1, ) = pair.getReserves();
            if (_r0 > r0 || _r1 > r1) {
```

```
                if (isSameToken0) {
                    r0 = _r0;
                    r1 = _r1;
                } else {
                    r0 = _r1;
                    r1 = _r0;
                }
                router = routers[i];
            }
        }
    }

    // (...SNIPPED...)

    function latestRoundData()
        external
        view
        returns (
            uint80 roundId,
            int256 answer,
            uint256 startedAt,
            uint256 updatedAt,
            uint80 answeredInRound
        )
    {
        return (0, _getAnswer(), 0, block.timestamp, 0);
    }

    function _getAnswer() private view returns (int256) {
        AggregatorV2V3Interface aggregator = AggregatorV2V3Interface(token0Agg);
        uint8 aggDecimals = aggregator.decimals();
        (, int256 answer, , , ) = aggregator.latestRoundData();
        (, uint112 r0, uint112 r1) = _selectRouterWithReserves();

        uint256 usdPrice = (r0 * uint256(answer) * (10 ** (18 - decimals0 + decimals1))) /
            (r1 * (10 ** aggDecimals));
        return int256(usdPrice);
    }
}
```

Listing 25.3 The *_queryRateUSD* function of the *PriceFeeds* contract

Recommendations

There is no recommendation code for this issue as it might break the contract functionality and require a decision from the *FWX* team in terms of business and protocol's core functionality.

However, when necessary to check the price difference between the two *DEXs*, we recommend the *FWX* team only utilize *DEX* pools with high liquidity to reduce the risk from price manipulation and other attack vectors from low liquidity.

Reassessment

The *FWX* team has acknowledged this issue with the statement as *"We acknowledged this issue and this case is acceptable for us since some tokens don't have oracle, such as COQ, so we decide to bypass price diff checking for these tokens"*.

No. 26	TODO Comments Should Be Resolved		
Risk	Medium	Likelihood	Medium
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/conditional/ConditionalFunc.sol* *contracts/src/core/CoreLiquidateWithoutSwap.sol* *contracts/src/gasless/ForwarderCore.sol*		
Locations	*ConditionalFunc._getMaxContractSize L: 380* *CoreLiquidateWithoutSwap._liquidatePositionWithoutSwap L: 334, 383* *ForwarderCore.repay L: 42*		

Detailed Issue

Before deploying contracts to the production, dev comments, especially "TODO" comments should be resolved. These comments indicate various unclear meanings such as:

- An uncompleted task that required the decision
- A missing functionality that should be implemented
- An uncompleted necessary validation check.

We found that the following "TODO" comment, needs to be resolved

- **Line 380 in the *ConditionalFunc* contract**
- **Lines 334 and 383 in the *CoreLiquidateWithoutSwap* contract**
- **Line 42 in the *ForwarderCore* contract**

Recommendations

We recommend the *FWX* team resolve the mentioned comment or remove the completed comment for readability and maintainability before bringing it to production.

Reassessment

The *FWX* team adopted our recommendation and fixed this issue.

No. 27	Insufficient Handling Of User's TPSLs When tpslTimesLimit Changes		
Risk	Medium	Likelihood	Medium
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/conditional/Conditional.sol* *contracts/src/conditional/ConditionalTPSL.sol* *contracts/src/conditional/ConditionalFunc.sol*		
Locations	*Conditional.setTpslTimesLimit L: 60 - 65* *ConditionalTPSL._setTPSL L: 100 - 112* *ConditionalTPSL._clearTPSL L: 114 - 122* *ConditionalFunc._assignTPSLMemoryToStorage L: 160 - 173*		

Detailed Issue

We found that when the *tpslTimesLimit* changes, the previous settings of the user's TPSLs will be unable to obtain the new *tpslTimesLimit* amount and/or cannot fully clear the previous setting if the *tpslTimesLimit* is decreased.

Consider the following scenario:

- Initial *tpslTimesLimit* = 5
- *TPSLs[0][0].length* = 5
- User A sets TPSL before the *tpslTimesLimit* changes and gets their *TPSLS[][].length* = 5

Scenario #1: ***tpslTimesLimit*** **is increased**

1. Only the initial TPSLs: *TPSLs[0][0]* is updated to the new *tpslTimesLimit* length: 7.
2. User A's TPSLs setting still contains the old *tpslTimesLimit* length: 5.
3. User A attempts to set their TPSLs again to obtain the new *tpslTimesLimit* via the *setTPSL* function by providing *_tpsl.length* = new_tpslTimesLimit: 7.
4. A transaction revert occurs in the *_assignTPSLMemoryToStorage* function as the array index is out of bound:
    a. *TPSLthisPos* contains the length: 5, bypassing the check of the initial new TPSLs: L167 in the code snippet 27.1.
    b. Loop assigns the TPSLs that loop through the given *_tpsl* (length: 7) and will revert as *TPSLthisPos[5]* is out of bound (L169 - 171 in the code snippet 27.1).

ConditionalFunc.sol

```
function _assignTPSLMemoryToStorage(
    uint256 nftID,
    bytes32 pairByte,
    TPSL[] memory _tpsl
) internal {
    TPSL[] storage TPSLthisPos = TPSLs[nftID][pairByte];
    if (TPSLthisPos.length == 0) {
        TPSLthisPos = TPSLs[0][0];
    }
    for (uint i = 0; i < _tpsl.length; i++) {
        TPSLthisPos[i] = _tpsl[i];
    }
    TPSLs[nftID][pairByte] = TPSLthisPos;
}
```

Listing 27.1 The *_assignTPSLMemoryToStorage* of the *ConditionalFunc* contract

Scenario #2: ***tpslTimesLimit* is decreased**

1. Only the initial TPSLs: *TPSLs[0][0]* is updated to the new *tpslTimesLimit* length: 3.
2. User A's TPSLs setting still contains the old *tpslTimesLimit* length: 5.
3. User A attempts to call the *clearTPSL* function to clear all their TPSLs (length: 5) but cannot, as the clear loop will only loop through the new *tpslTimesLimit*: 3, leaving the rest uncleared.

ConditionalFunc.sol

```
function _clearTPSL(uint256 nftID, bytes32 pairByte) internal {
    TPSL[] storage _tpsls = TPSLs[nftID][pairByte];
    TPSL[] memory oldTPSL = _tpsls;
    for (uint8 i = 0; i < tpslTimesLimit; i++) {
        if (i < _tpsls.length) delete _tpsls[i];
    }
    // cant not assigned by replace TPSLs[0][0] cuz it's not change storage
    emit SetTPSL(msg.sender, nftID, pairByte, oldTPSL, _tpsls);
}
```

Listing 27.2 The *_clearTPSL* of the *ConditionalTPSL* contract

Recommendations

We recommend updating the logic to properly handle both increased and decreased *tpslTimesLimit* values to ensure consistent and accurate behavior when *tpslTimesLimit* changes.

Reassessment

The *FWX* team updates the *_clearTPSL* and *_assignTPSLMemoryToStorage* functions to fix this issue and confirms that the current code functions as designed, with the following behaviors:

- The initial *TPSLs* length will be equal to *initialTPLSLength* at the time of initialization.
- Once initialized, the *TPSLs* length will not change.
- The *trigger* function of *ConditionalTPSL* contract will execute the array of a given *TPSLs*, regardless of any changes to the *initialTPLSLength.*

No. 28	Trigger Order Fee For Suddenly Open Order Is Kept In Conditional Contract		
Risk	Medium	Likelihood	Medium
		Impact	Medium
Functionality is in use	In use	Status	Acknowledged
Associated Files	*contracts/src/conditional/ConditionalOrder.sol*		
Locations	*ConditionalOrder._placeOrder L: 127*		

Detailed Issue

We found that when a user places an order via the *_placeOrder* function of the *ConditionalOrder* contract and the order is opened suddenly, the *triggerOrderTxServiceCharge* that should typically be transferred to the triggerer is instead transferred to the *Conditional* contract. This is potentially unintended behavior.

Moreover, the fee is not stuck in the *Conditional* contract as the *Conditional* contract manager can call the *ownerApprove* function to retrieve that fee.

ConditionalOrder.sol

```
76   function _placeOrder(PlaceOrderStruct memory params) internal {

             // (...SNIPPED...)

119          {
120          uint256 upperEdge = (params.targetPrice * (WEI_PERCENT_UNIT +
     params.slipPage)) /
121              WEI_PERCENT_UNIT;
122          uint256 lowerEdge = (params.targetPrice * (WEI_PERCENT_UNIT -
     params.slipPage)) /
123              WEI_PERCENT_UNIT;
124
125          // open position if the price range is acceptable otherwise place limit
     order
126          if (newIsLong ? upperEdge >= rate : lowerEdge <= rate) {
127              _openPositionToPool(nftId, rate, newOrder);
128              return;
129          } else {
```

Listing 28.1 The *_placeOrder* function of the *ConditionalOrder* contract

CoreFutureBaseFunc.sol

```
function _conditionalExecutionFeeHandler(
    address caller,
    address collateralToken,
    uint256 serviceCharge,
    uint256 nftId,
    bytes32 pairByte
) internal {
    if (serviceCharge > 0) {
        (uint256 rate, ) = _queryRateUSD(wethAddress);
        rate = (rate * tokenPrecisionUnit[collateralToken]) /
tokenPrecisionUnit[wethAddress];
        serviceCharge = (rate * serviceCharge) / WEI_UNIT;

        uint256 wallet = wallets[nftId][pairByte];
        _updateWallet(nftId, pairByte, wallet - serviceCharge);

        serviceCharge = serviceCharge / 2;
        _settleAndTransferFutureTradeFee(collateralToken, 0, serviceCharge);
        _transferOut(caller, collateralToken, serviceCharge);
    }
}
```

Listing 28.2 The *_conditionalExecutionFeeHandler* function of the *ConditionalOrder* contract

Recommendations

We recommend that the team decide on the logic for handling the returned fee for this specific case in alignment with the business requirements.

Reassessment

The *FWX* team has acknowledged this issue as it is an intentional design.

No. 29	Inadequate Handling Of Paused Or Unsupported Price Feeds In _placeOrder Function		
Risk	Low	Likelihood	Low
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/conditional/ConditionalOrder.sol*		
Locations	*ConditionalOrder._placeOrder L: 114*		

Detailed Issue

The *_placeOrder* function in the *ConditionalOrder* contract does not appropriately handle scenarios where the global price feed is paused or lacks support for the given tokens. This results in unintended consequences for users when placing orders:

This oversight results in unintended consequences for users when placing orders:

Long Orders: Users can place long orders, which can trigger suddenly even when the global price feed is paused. Although the transaction eventually reverts during the price difference check within the open position logic, users incur unnecessary gas costs.

Short Orders: Users can successfully place short orders as limit orders, unaffected by the paused global price feed.

ConditionalOrder.sol

```
76  function _placeOrder(PlaceOrderStruct memory params) internal {

        // (...SNIPPED...)

114     uint256 rate = _getRateInCollaUnit(
115         newIsLong ? params.swapTokenAddress : params.borrowTokenAddress,
116         params.collateralTokenAddress
117     );
118
119 {
120     uint256 upperEdge = (params.targetPrice * (WEI_PERCENT_UNIT +
    params.slipPage)) /
121         WEI_PERCENT_UNIT;
122     uint256 lowerEdge = (params.targetPrice * (WEI_PERCENT_UNIT -
    params.slipPage)) /
```

```
        WEI_PERCENT_UNIT;

    // open position if the price range is acceptable otherwise place limit order
    if (newIsLong ? upperEdge >= rate : lowerEdge <= rate) {
        _openPositionToPool(nftId, rate, newOrder);
        return;
    } else {
            if (rate < params.targetPrice) {
                newOrder.lowerTargetPrice = params.targetPrice;
                newOrder.upperTargetPrice = upperEdge;
            } else {
                newOrder.lowerTargetPrice = lowerEdge;
                newOrder.upperTargetPrice = params.targetPrice;
            }

            nftIdOrder.push(newOrder);
            uint256 currentIndex = nftIdOrder.length - 1;

            emit PlaceOrder(msg.sender, newOrder, uint8(currentIndex));

            _triggerOrder(nftId, pairByte, currentIndex);
        }
    }
}
```

Listing 29.1 The *_placeOrder* function of the *ConditionalOrder* contract

Recommendations

We recommend handling the return of the price feed rate before performing any subsequent logic in the *_placeOrder* function of the *ConditionalOrder* contract.

This ensures the proper handling of scenarios where the global price feed is paused or does not support the given tokens, preventing unintended consequences for users.

Reassessment

The *FWX* team adopted our recommendation and fixed this issue.

No. 30	Inconsistent Router Interface Usage In ConditionalFunc And CoreSwappingV3 Contracts		
Risk	Low	Likelihood	Low
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/conditional/ConditionalFunc.sol*		
Locations	*Several functions in ConditionalFunc contract*		

Detailed Issue

We identified an inconsistency in the router interfaces used between the *ConditionalFunc* and *CoreSwappingV3* contracts:

- In the *ConditionalFunc* contract, the router interface is represented by the *Forward* interfaces: *IForwardRouter02*, *IForwardFactory*, and *IForwardPair*.
- In the *CoreSwappingV3* contract, the router interface uses multiple interfaces, including *IXlipless* (an internal interface) and *IRouter* (global router interfaces).

For instance, the *_getSwapFeeRate* function in the *ConditionalFunc* contract uses the *IForwardRouter02* interface to interact with the router at index 0, whereas the *CoreSwappingV3* contract uses the *IXlipless* interface for the same purpose.

ConditionalFunc.sol

```
function _getSwapFeeRate(
    uint256 routerIndex,
    address token0,
    address token1
) internal view returns (uint256 swapFeeRate) {
    IAPHCore core = _getCoreProxy();

    if (routerIndex == 0) {
        IForwardPair pair = IForwardPair(_getPair(0, token0, token1));
        swapFeeRate = (uint256(pair.fee()) * WEI_UNIT) / 100;
    } else {
        // Other router's swap fee rate
        swapFeeRate = core.swapFeeRates(core.routers(routerIndex));
    }
}
```

Listing 30.1 The *_getSwapFeeRate* function of the *ConditionalFunc* contract

CoreSwappingV3.sol

```
function _validateRouter(
    ValidateRouterArgs memory args
) internal view returns (uint256[] memory amounts, uint256 swapFee,
ValidateError err) {
    amounts = new uint[](2);
    Pair memory pair = pairs[args.pairByte];
    SwapConfig memory cfg =
swapConfigs[routers[args.routerIndex]][args.pairByte];
    (uint256[] memory _amounts, uint256 _swapFee) = _getAmounts(
        args.isExactOutput,
        true,
        args.routerIndex,
        args.amountInput,
        args.path
    );

    // conditions for internal DEX
    if (args.routerIndex == 0) {
        {
            // validate token whitelisted
            IXlipless xlipless = IXlipless(routers[0]);
```

Listing 30.2 The *_validateRouter* function of the *CoreSwappingV3* contract

Recommendations

We recommend standardizing the router interface usage across the *ConditionalFunc* and *CoreSwappingV3* contracts.

Reassessment

The *FWX* team adopted our recommendation and fixed this issue by using the same router interface.

ConditionalFunc.sol

```
function _getSwapFeeRate(
    uint256 routerIndex,
    address token0,
    address token1
) internal view returns (uint256 swapFeeRate) {
    IAPHCore core = _getCoreProxy();

```

```
    if (routerIndex == 0) {
        // Xlipless swap fee rate
        swapFeeRate = IXlipless(core.getRouters()[0]).fees(token0, token1);
    } else {
        // Other router's swap fee rate
        swapFeeRate = core.swapFeeRates(core.getRouters()[routerIndex]);
    }
}
```

Listing 30.3 The updated *_getSwapFeeRate* function of the *ConditionalFunc* contract

No. 31	Recommended Following Best Practices For Upgradeable Smart Contracts		
Risk	Low	Likelihood	Low
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/core/APHCore.sol* *contracts/src/pool/APHPool.sol* *contracts/src/pool/APHPoolV2.sol* *contracts/src/gasless/Forwarder.sol* *contracts/src/conditional/Conditional.sol*		
Locations	*APHCore.constructor L: 12* *APHPool.constructor L: 12* *APHPoolV2.constructor L: 13* *Forwarder.constructor L: 24* *Conditional.constructor L: 8*		

Detailed Issue

The following contracts should enhance the disable initializer mechanism to be broadly supported in future upgrades and follow the best practices.

- The *APHCore* contract
- The *APHPool* contract
- The *APHPoolV2* contract
- The *Forwarder* contract
- The *Conditional* contract

The practice below performs equivalent to *reinitializer(1)* which does not protect in the case of the contract upgrades that require reinitialization of the next version (version > 1).

APHCore.sol

```
// (...SNIPPED...)
contract APHCore is APHCoreProxy, APHCoreSettingProxy, CoreEvent,
CoreSettingEvent {
    constructor() initializer {}
// (...SNIPPED...)
}
```

Listing 31.1 The *constructor* of the *APHCore* contract

Recommendations

We recommend revising to use the *_disableInitializers* function.

The *_disableInitializers* function guards against future reinitializations by setting _initialized version to the max supported version (uint8.max for OpenZeppelin contract version <= v4.9.5, uint64.max, >= v5.0.0, for OpenZeppelin contract version).

APHCore.sol

```
// (...SNIPPED...)
contract APHCore is APHCoreProxy, APHCoreSettingProxy, CoreEvent, CoreSettingEvent {
    constructor() {
        _disableInitializers();
    }

// (...SNIPPED...)
}
```

Listing 31.2 The improved *constructor* of the *APHCore* contract

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *FWX* team adopted our recommendation and fixed this issue.

No. 32	Unsafe ABI Encoding		
Risk	Low	Likelihood	Low
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/core/APHCoreProxy.sol* *contracts/src/core/APHCoreSettingProxy.sol* *contracts/src/conditional/ConditionalTPSL.sol* *contracts/src/conditional/ConditionalProxy.sol* *contracts/src/gasless/SmartWallet.sol*		
Locations	*APHCoreProxy L: 128, 149, 191, 205* *APHCoreSettingProxy L: 296, 310, 323* *ConditionalProxy L: 11, 17, 37, 54, 64, 73* *ConditionalTPSL L: 77* *SmartWallet L: 119*		

Detailed Issue

We found that the use of ***abi.encodeWithSignature*** and ***abi.encodeWithSelector*** functions for generating calldata in low-level calls introduce potential risks in several functions.

The first function is susceptible to typographical errors, and the second lacks type safety. These vulnerabilities can lead to unexpected and unsafe outcomes in smart contract operations.

APHCoreProxy.sol

```
199 function openPosition(
200      APHLibrary.OpenPositionParams memory params,
201      APHLibrary.TokenAddressParams memory addressParams,
202      address conditionalTradingAddressFromPool
203 ) external {
204      // TODO: change to encodeCall
205      bytes memory data = abi.encodeWithSignature(
206 "openPosition((uint256,uint256,uint256,uint256,uint256,uint256,bool),(address,address,address),address)",
207          params,
208          addressParams,
209          conditionalTradingAddressFromPool
210      );
211      _delegateCall(coreFutureOpeningAddress, data);
```

```
}
```

Listing 32.1 The *openPosition* function of the *APHCoreProxy* contract

Recommendations

We recommend replacing any instances of unsafe ABI encodings with ***abi.encodeCall***, which verifies that the given values match the types anticipated by the called function while avoiding typographical errors.

Reference from docs.soliditylang.org
abi.encodeCall(function functionPointer, (...)) returns (bytes memory): ABI-encodes a call to functionPointer with the arguments found in the tuple. Performs a full type-check, ensuring the types match the function signature.

Reassessment

The *FWX* team adopted our recommendation and fixed this issue.

No. 33	Recommended Event Emissions For Transparency And Traceability		
Risk	Informational	Likelihood	Low
		Impact	Low
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/conditional/Conditional.sol*		
Locations	*Conditional.setTpslTimesLimit L: 60 - 66*		

Detailed Issue

We consider operations of the following state-changing function important and require proper event emissions for improving transparency and traceability:

Conditional.sol

```
function setTpslTimesLimit(uint8 newTimesLimit) external
onlyAddressTimelockManager {
    // uint8 oldTpslTimesLimit = tpslTimesLimit;
    _initialTPSL(newTimesLimit);
    tpslTimesLimit = newTimesLimit;
    // emit SetTpslTimesLimit(msg.sender, oldTpslTimesLimit, newTimesLimit);
}
```

Listing 33.1 The *setTpslTimesLimit* function of the *Conditional* contract

Recommendations

We recommend emitting relevant events on the associated functions to improve transparency and traceability.

Reassessment

The *FWX* team adopted our recommendation and fixed this issue.

No. 34	Incorrectly Emitted Event Value		
Risk	Informational	Likelihood	Low
		Impact	Low
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/conditional/Conditional.sol* *contracts/src/core/CoreFutureWallet.sol*		
Locations	*Conditional.setOrderTimesLimit L: 68* *CoreFutureWallet L: 67, 75, 136, 139*		

Detailed Issue

We found that the incorrect event emissions as shown in the listed below

1. The *oldOrderTimesLimit* value in the *Conditional.setOrderTimesLimit* function
2. The *msg.sender* value incorrectly represents the owner of the NFT if the caller is a Forwarder.

Conditional.sol

```
function setOrderTimesLimit(uint8 newTimesLimit) external
onlyAddressTimelockManager {
     uint8 oldOrderTimesLimit = tpslTimesLimit;
     orderTimesLimit = newTimesLimit;
     emit SetOrderTimesLimit(msg.sender, oldOrderTimesLimit, newTimesLimit);
}
```

Listing 34.1 The incorrect *oldOrderTimeLimit* in the *setOrderTimesLimit* function

CoreFutureWallet.sol

```
function _depositCollateral(
        uint256 nftId,
        address collateralTokenAddress,
        address underlyingTokenAddress,
        uint256 amount
    ) internal returns (uint256) {
    // (...SNIPPED...)
    emit UpdateWallet(
            msg.sender,
```

```
            nftId,
            pairByte,
            wallets[nftId][pairByte] - amount,
           wallets[nftId][pairByte]
      );

        emit DepositCollateral(
            msg.sender,
            nftId,
            collateralTokenAddress,
            underlyingTokenAddress,
            pairByte,
            amount
        );
        // (...SNIPPED...)
}

function _withdrawCollateral(
        uint256 nftId,
        address collateralTokenAddress,
        address underlyingTokenAddress,
        uint256 amount
    ) internal returns (uint256) {
    // (...SNIPPED...)
    emit UpdateWallet(msg.sender, nftId, pairByte, wallet,
wallets[nftId][pairByte]);

    emit WithdrawCollateral(
        msg.sender,
        nftId,
        collateralTokenAddress,
        underlyingTokenAddress,
        pairByte,
        amount
    );
    return amount;
}
```

Listing 34.2 The incorrect NFT owner value in the *_depositCollateral and _withdrawCollateral* functions

Recommendations

We recommend revising the mentioned incorrect event emission to improve the transparency and traceability of the protocol.

Reassessment

The *FWX* team adopted our recommendation and fixed this issue.

No. 35	Recommended Removing Unused Code		
Risk	Informational	Likelihood	Low
		Impact	Low
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/conditional/Conditional.sol* *contracts/src/conditional/ConditionalFunc.sol* *contracts/src/conditional/ConditionalOrder.sol* *contracts/src/conditional/ConditionalTPSL.sol* *contracts/src/core/CoreFutureClosing.sol* *contracts/src/gasless/ForwarderFunc.sol* *contracts/src/gasless/SmartWallet.sol*		
Locations	*Conditional L: 61, 64* *ConditionalFunc L: 374, 375* *ConditionalOrder L: 10* *ConditionalTPSL L: 103* *CoreFutureClosing L: 6* *ForwarderFunc L: 111* *SmartWallet L: 184*		

Detailed Issue

We found that unused codes can be removed for readability and maintainability as listed below.

- The *Conditional* contract line 61, 64
- The *ConditionalFunc* contract line 374, 375
- The *ConditionalOrder* contract line 10
- The *ConditionalTPSL* contract line 103
- The *CoreFutureClosing* contract line 6
- The *ForwarderFunc* contract line 111
- The *SmartWallet* contract line 184

Recommendations

We recommend removing the unused codes to improve readability and maintainability of the protocol.

Reassessment

The *FWX* team adopted our recommendation and fixed this issue.

No. 36	Inconsistent Usage Of Manager Roles		
Risk	Informational	Likelihood	Low
		Impact	Low
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/core/CoreSetting.sol* *contracts/src/conditional/Conditional.sol*		
Locations	*CoreSetting.setForwTradingVaultAddress L: 39* *Conditional.setTpslTimesLimit L: 60* *Conditional.setOrderTimesLimit L: 67*		

Detailed Issue

We found inconsistencies in the usage of the manager roles. Specifically, the *onlyAddressTimelockManager* role should be used for setting up address states in the contract, while the *onlyConfigTimelockManager* role should be used for setting up system configuration states. Below is a list of functions that need role adjustments to ensure proper usage:

1. The *CoreSetting.setForwTradingVaultAddress* function should be changed to *onlyAddressTimelockManager*.
2. The *Conditional.setTpslTimesLimit* function should be changed to *onlyConfigTimelockManager*.
3. The *Conditional.setOrderTimesLimit* function should be changed to *onlyConfigTimelockManager*.

CoreSetting.sol

```
28 function setForwLendingVaultAddress(address _address) external
   onlyAddressTimelockManager {
29         address oldAddress = forwLendingVaultAddress;
30         forwLendingVaultAddress = _address;
31 
32         if (oldAddress != address(0)) {
33             IERC20Upgradeable(forwAddress).safeApprove(oldAddress, 0);
34         }
35         IERC20Upgradeable(forwAddress).safeApprove(forwLendingVaultAddress,
   type(uint256).max);
36         emit SetForwLendingVaultAddress(msg.sender, oldAddress, _address);
37     }
38 
39 function setForwTradingVaultAddress(address _address) external
   onlyConfigTimelockManager {
```

```
    address oldAddress = forwTradingVaultAddress;
    forwTradingVaultAddress = _address;

    emit SetForwTradingVaultAddress(msg.sender, oldAddress, _address);
}
```

Listing 36.1 The *setForwLendingVaultAddress* and *setForwTradingVaultAddress* functions of the *CoreSetting* contract

Recommendations

We recommend updating the roles for these functions to ensure they align with the intended usage, enhancing the security and maintainability of the contract.

Reassessment

The *FWX* team adopted our recommendation and fixed this issue.

No. 37	Misspellings And Typos In Codebase		
Risk	Informational	Likelihood	Low
		Impact	Low
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/utils/TransperantProxy.sol* *contracts/src/gasless/ForwarderFunc.sol*		
Locations	*TransperantProxy.sol* *ForwarderFunc.sol L: 20, 25, 49, 53*		

Detailed Issue

We found several misspellings and typos that need correction:

1. The filename *TransperantProxy.sol* should be corrected.
2. The function name *ForwaderFunc.isOnylExecutor* line 49 should be corrected.
3. The function name *ForwaderFunc.isOnylExecutionToken* line 53 should be corrected.
4. The modifier name *onylExecutor* in ForwaderFunc contract line 20 should be corrected.
5. The modifier name *onylExecutionToken* in ForwaderFunc contract line 25 should be corrected.

Recommendations

We recommend correcting the misspellings and typos to ensure clarity and maintainability of the code.

Reassessment

The *FWX* team adopted our recommendation and fixed this issue.

No. 38	Out Of Audit Scope		
Risk	Informational	Likelihood	Low
		Impact	Low
Functionality is in use	In use	Status	Acknowledged
Associated Files	*contracts/src/core/CoreSwappingV3.sol* *contracts/src/conditional/ConditionalFunc.sol* *contracts/src/core/CoreLiquidateWithoutSwap.sol*		
Locations	*CoreSwappingV3._getSwapFeeRate L: 599* *CoreSwappingV3._validateRouter L: 477* *ConditionalFunc._getSwapFeeRate L: 85* *ConditionalFunc._getFutureMarginFromPosition L: 328* *ConditionalFunc_getMaxContractSize L: 347, 372* *CoreLiquidateWithoutSwap._setPositionAndCheckIncreaseBalance L: 587*		

Detailed Issue

We found several functions that make external calls to interfaces that are out of the audit scope. These calls must work properly and be secure. Below is the list of interfaces and their usage:

1. The *IXlipless* interface used in the *CoreSwappingV3* contract for calculating fees, checking token whitelist, verifying allowed paths, getting max swap sizes, etc.
2. The *IHelperFutureTrade* interface is used in the *ConditionalFunc* contract for retrieving margin after opening a position, getting trader balance, getting exit price, etc.
3. The *IAPHCoreLiquidateCallee* interface used in the *CoreLiquidateWithoutSwap* contract for liquidating positions without swaps, liquidating loans without swaps, etc.

These external calls should be audited to ensure the security of the implementation contracts.

CoreSwappingV3.sol

```
592 function _getSwapFeeRate(
593      uint256 routerIndex,
594      address token0,
595      address token1
596  ) internal view returns (uint256 swapFeeRate) {
597      if (routerIndex == 0) {
598          // Xlipless swap fee rate
599          swapFeeRate = IXlipless(routers[0]).fees(token0, token1);
```

```
    } else {
        // Other router's swap fee rate
        swapFeeRate = swapFeeRates[routers[routerIndex]];
    }
}
```

Listing 38.1 The *_getSwapFeeRate* function of the *CoreSwappingV3* contract

Recommendations

We recommend conducting a comprehensive audit of the implementation contracts for these interfaces to ensure they operate securely and as expected.

Reassessment

The *FWX* team has acknowledged this issue.

Appendix

About Us

Founded in 2020, Valix Consulting is a blockchain and smart contract security firm offering a wide range of cybersecurity consulting services such as blockchain and smart contract security consulting, smart contract security review, and smart contract security audit.

Our team members are passionate cybersecurity professionals and researchers in the areas of private and public blockchain technology, smart contract, and decentralized application (DApp).

We provide a service for assessing and certifying the security of smart contracts. Our service also includes recommendations on smart contracts' security and gas optimization to bring the most benefit to users and platform creators.

Contact Information

info@valix.io

https://www.facebook.com/ValixConsulting

https://twitter.com/ValixConsulting

https://medium.com/valixconsulting

References

Title	Link
OWASP Risk Rating Methodology	https://owasp.org/www-community/OWASP_Risk_Rating_Methodology
Smart Contract Weakness Classification and Test Cases	https://swcregistry.io/